Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tilen Faganel

Ogrodje za razvoj mikrostoritev v Javi in njihovo skaliranje v oblaku

diplomsko delo

univerzitetni študijski program računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite arhitekturo mikrostoritev in jo primerjajte s klasičnimi in storitvenimi arhitekturami. Analizirajte koncepte in vzorce razvoja mikrostoritev v Javi. Identificirajte omejitve obstoječih tehnologij in platform pri uporabi arhitekture mikrostoritev. Pripravite zasnovo lastnega ogrodja za razvoj mikrostoritev na platformi Java EE. Na vzorčni implementaciji ogrodja prikažite delovanje ogrodja, na primeru pa prednosti in morebitne slabosti uporabe mikrostoritev v konkretnih aplikacijah. Analizirajte možnosti skaliranja mikrostoritev v oblaku ter na primeru demonstrirajte možnosti skaliranja z uporabo virtualizacije Docker.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Tilen Faganel, z vpisno številko **63110215**, sem avtor diplomskega dela z naslovom:

Ogrodje za razvoj mikrostoritev v Javi in njihovo skaliranje v oblaku

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 13. septembra 2015 Podpis avtorja:

Zahvaljujem se mentorju prof. dr. Matjažu Branku Juriču za usmerjanje in podporo ter dr. Sebastijanu Špragerju za pomoč pri izdelavi diplomske naloge. Zahvaljujem se tudi Nataši Bufon Jakončič za pomoč pri lektoriranju in oblikovanju naloge. Zahvaljujem se še vsem mojim prijateljem in celotni družini za vse zabavne čase in podporo tekom mojega študija.

It is not the destination that matters — it is the journey.

Kazalo

Slike

Odseki

Seznam uporabljenih kratic

IT	Information Technology
API	Application Programming Interface
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
REST	Representational State Transfer
XML	Extensible Markup Language
JSON	JavaScript Object Notation
WSDL	Web Service Definition Language
WADL	Web Application Definition Language
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
NoSQL	Non Structured Query Language
HTTP	Hyper-Text Transfer Protocol
HTML	Hyper-Text Markup Language
ORM	Object-Relational Mapping
DI	Dependency Injection
CDN	Content Delivery Network
JSR	Java Specification Request
JAR	Java Archive
WAR	Web Application Archive
EAR	Enterprise Archive
IDE	Integrated Development Environment

JPA	Java Persistence API
CDI	Context and Dependency Injection
EJB	Enterprise Java Beans
JTA	Java Transaction API
JAX-RS	Java API for Restful Web Services
JAX-WS	Java API for XML Web Services
JMS	Java Message Service
JSP	Java Server Pages
JSF	Java Server Faces
EL	Expression Language
JSON-P	Java API for Json Processing
JVM	Java Virtual Machine
JNDI	Java Naming and Directory Interface

Povzetek

Uporaba spletnih aplikacij v zadnjih letih strmo raste, zato je način arhitekturne zasnove in razvoja takih aplikacij postal vedno pomembnejši. Arhitektura mikrostoritev naslovi potrebe sodobnih poslovnih aplikacij, ki imajo veliko število uporabnikov in se razvijajo iterativno. Osredotoča se na dekompozicijo aplikacij v manjše obvladljive funkcionalno zaključene storitve, ki jih neodvisno in samostojno upravljamo ter namestimo. Kljub temu razvoj pravih mikrostoritev v Javi EE s trenutnimi orodji ni mogoč. V ta namen smo v diplomski nalogi razvili ogrodje za razvoj mikrostoritev s pomočjo tehnologij Java EE. Omogoča nam, da izberemo komponente, ki jih nato ogrodje ustrezno inicializira in zapakira v samostojen arhiv, ki ga lahko poganjamo neodvisno brez zunanjih odvisnosti. Prav tako pa nam močno olajša namestitev in skaliranje razvitih mikrostoritev v oblačna okolja PaaS. Ključni doprinos diplomskega dela je razvoj izvirnega ogrodja za razvoj mikrostoritev v Javi EE, ki je prvo tako ogrodje za platformo Java in omogoča enostaven in hiter razvoj mikrostoritev.

Ključne besede: mikrostoritve, skalabilnost, oblak, spletne aplikacije, java ee, docker.

Abstract

In recent years' usage of web applications has increased. Therefore, it has become increasingly important how these applications are designed and developed. The microservice architecture addresses the needs of modern business applications with a large number of users and iterative development. The architecture focuses on decomposing applications into small autonomous services which are deployed and managed independently. However, developing true microservices in Java EE is not possible with the currently available tools. In this thesis we designed and implemented a framework for developing microservices with stock Java EE technologies. The framework allows us to select the required Java EE components, which are then bootstrapped and packaged into an executable archive that has no external dependencies. It also greatly simplifies deploying and scaling developed microservices into various PaaS and other cloud environments. The key contribution of this thesis is an original framework for developing microservices using Java EE. It is the first such framework for the Java platform and enables easy and rapid development of microservices.

Keywords: microservices, scalability, cloud, web applications, java ee, docker.

Poglavje 1

Uvod

Spletne aplikacije in storitve so danes ključen del našega vsakdanjega življenja. Priča smo ogromni rasti vseh aspektov spleta, ki poleg končnih produktov, vključuje tudi orodja za izdelavo slednjih. Zasledimo lahko vedno bolj dinamične in bogate spletne aplikacije, hkrati pa morajo te biti vedno bolj fleksibilne in prilagodljive, tako da jih lahko uporabljamo na vedno večjem številu raznolikih naprav (mobilni telefoni, pametne ure, tablice in druge naprave). Če upoštevamo še ogromno rast uporabnikov interneta [1], morajo spletne aplikacije, poleg naštetega, biti sposobne obravnavati veliko število uporabnikov.

Za lažji in učinkovitejši razvoj takih spletnih aplikacij se je v zadnjih letih pojavilo veliko število novih tehnologij, ki uporabljajo številne različne programske jezike. Tako lahko danes razvijamo neko spletno aplikacijo s pomočjo objektno orientiranih programskih jezikov (C++, Java, C#, ...) in vse do funkcijskih programskih jezikov (Scala, Haskell, ...), kar omogoča izjemno fleksibilnost. Določeni jeziki so boljši za nekatere specifične zadeve, npr. obdelavo velikega števila podatkov, nekateri pa za druge, npr. obdelavo velikega števila paralelnih zahtevkov. Kljub številni izbiri se danes v večjih organizacijah še vedno največ uporablja Java, natančneje Java EE. Glavni razlog je veliko število knjižnic in orodij, dobra komercialna podpora velikih podjetij, veliko število razpoložljivega kadra, uveljavljena infrastruktura certificiranja

kadra, relativno standarizirani načini razvoja in dobra kompatibilnost za starejše aplikacije. Novejše tehnologije so bolj priljubljene pri "startup-ih". Ne glede na to, katero tehnologijo oz. programski jezik izberemo, moramo aplikacijo zasnovati in nato razviti skladno z izbrano arhitekturo.

Če se osredotočimo na poslovne aplikacije večjih organizacij narejene v Javi EE, zasledimo, da take aplikacije večinoma uporabljajo monolitno arhitekturo, k kateri zapakiramo celotno aplikacijo v en EAR (*Enterprise Application Archive*) ali WAR (*Web Application Archive*) arhiv, ki ga nato namestimo na aplikacijski strežnik. Arhiv vsebuje vse module aplikacije, ki so potrebni za delovanje (npr. modul za komunikacijo s podatkovno bazo in modul za poslovno logiko). WAR pogosto vsebuje modul, ki je odgovoren za prikaz spletne aplikacije, medtem ko EAR vsebuje še vse preostale module. Čeprav ima ta pristop določene prednosti, predvsem z vidika enostavnosti razvoja, je aplikacijo težje vzdrževati in - še posebej pomembno - težje skalirati v oblačnih infrastrukturah.

Arhitektura mikrostoritev naslavlja te probleme in sicer tako, da razbije veliko aplikacijo na več manjših mikrostoritev, ki jih nato neodvisno in samostojno razvijamo, namestimo, upravljamo in skaliramo. Vsaka mikrostoritev ima s tem namenom točno definiran sklop funkcionalnosti, ki jih upravlja, prav tako pa vsebuje vmesnik za komunikacijo med različnimi mikrostoritvami. Omenjen način prinese veliko fleksibilnosti pri izbiri tehnologij, posodabljanja in upravljanja posameznih mikrostoritev oz. delov aplikacije. V primeru Jave EE je vsaka mikrostoritev zapakirana v svoj neodvisen JAR (*Java Archive*) arhiv, ki ga lahko samostojno zaženemo. Opisani pristop ima tudi svoje slabosti. Ker razbijemo aplikacijo na več delov, se poveča kompleksnost pri namestitvi, konfiguraciji in pakiranju celotne aplikacije ter njenih odvisnosti.

V diplomski nalogi smo podrobno razčlenili arhitekturo mikrostoritev in njene razlike glede na obstoječo uveljavljeno monolitno arhitekturo. Predstavili smo, kako nam mikrostoritve pomagajo pri namestitvi in poganjanju aplikacij v oblačnih infrastrukturah, kar je danes za zadostno delovanje skoraj

nujno. Pregledali smo obstoječa orodja in podporo za razvoj mikrostoritev v Javi EE in kaj nam taka orodja ponujajo glede na željene funkcionalnosti. Nazadnje smo zasnovali in razvili svoje ogrodje za razvoj mikrostoritev z obstoječimi Java EE tehnologijami, ki naslavlja in rešuje predstavljene probleme arhitekture. Preostala orodja na trgu bodisi ne uporabljajo Jave EE bodisi samo majhen del, kar pomeni, da ne morejo izkoristiti vseh orodij in knjižnic, ki jih ponuja, in posledično ne pridejo v upoštev pri organizacijah, ki trenutno uporabljajo Javo EE. Ogrodje omogoča, da Java EE tehnologije poljubno izberemo in preprosto konfiguriramo ter zapakiramo v samostojen arhiv, ki ga lahko nato poženemo kjerkoli, kjer imamo nameščeno javansko izvajalno okolje. Takšen način omogoča nov, modern in učinkovitejši razvoj Java EE aplikacij, ki so primerne za današnji svet aplikacij, ki jih uporablja vedno večje število uporabnikov, hkrati pa ohranja vse obstoječe Java EE tehnologije. Ogrodje predstavlja dobro ravnovesje med modernimi in obstoječimi uveljavljenimi pristopi razvoja ter olajša uporabo Jave EE v oblačni infrastrukturi.

Diplomska naloga je razčlenjena na 6 poglavij. V poglavju 2 je podrobno opisana arhitektura mikrostoritev, kakšne so njene razlike v primerjavi z monolitno arhitekturo in njena relacija s SOA (Service Oriented architecture). V poglavju 3 opišemo koncepte in vzorce za razvoj mikrostoritev v Javi. Poglavje 4 opisuje lastno ogrodje za razvoj mikrostoritev v Javi EE, kaj omogoča, kako je zasnovan, podamo pa tudi primer preproste uporabe pri zasnovi enostavne spletne aplikacije. V poglavju 5 prikažemo, kako lahko s pomočjo ogrodja preprosto upravljamo, konfiguriramo in skaliramo aplikacijo v oblačni infrastrukturi, ki je osnovana na tehnologiji Docker. V poglavju 6 podamo sklep diplomskega dela.

Poglavje 2

Arhitektura mikrostoritev

Zadnja leta smo priča veliki rasti spletnih aplikacij in storitev. Število uporabnikov strmo raste (npr. Facebook ima že preko 1 milijarde uporabnikov [2]), ti pa pričakujejo bogato, interaktivno in dinamično uporabniško izkušnjo ne glede na to s kakšnim medijem bodo do neke aplikacije dostopali bodisi na brskalniku svojega prenosnega računalnika bodisi na svojem mobilnem telefonu ali tablici. V vsakem primeru morajo aplikacije obvladovati veliko število različnih uporabnikov, ki dostopajo na različne načine. V primeru, da jim to ne uspe (npr. v primeru, ko aplikacija ni dovolj odzivna, ni zmožna obdelovati večjega števila sočasnih uporabnikov ipd.), bodo uporabniki sprejeli druge konkurenčne storitve, ki jim omogočajo boljšo izkušnjo.

Da bi zagotovili vedno večje število uporabnikov, morajo biti aplikacije zasnovane z visoko skalabilnostjo. Tako lahko aplikacija ustrezno raste vzporedno z rastjo uporabnikov. Prav tako morajo aplikacije biti vse visoko razpoložljive, saj lahko vsaka prekinitev v delovanju ponudnika stane veliko denarja in tudi uporabnikov. Aplikacija mora delovati tudi na raznih oblačnih infrastrukturah, ki močno olajšajo vzdrževanje in skaliranje, kar pa je pri velikih sistemih še kako pomembno. Poleg vsega pripomorejo k zmanjšanju skupnih obratovalnih stroškov.

Ponudniki pogosto želijo objavljati posodobitve in popravke, v nekaterih primerih celo večkrat dnevno. Ob današnji veliki rasti trga in vedno

večji konkurenci je pogosto posodabljanje skorajda obvezno, saj je zelo pomembno, da ponudniki ohranijo svojo konkurenčno prednost oz. se približajo konkurentom. Tudi v primeru pojave hroščev jim je v velikem interesu, da to čimprej in brez velikih težav odpravijo in omogočijo svojim uprabnikom najboljšo izkušnjo. Prav tako morajo poskrbeti, da so vsi deli aplikacije ustrezno integrirani in da pri posodobitvah ne pride do problemov s kompatibilnostjo. V nasprotnem primeru se lahko poruši delovanje celotne aplikacije.

V zakup vsemu naštetemu ni več primerno oz. celo efektivno razvijati spletne aplikacije, ki ustreza monolitni arhitekturi. Takšen način razvoja je bil populariziran v času, ko je bil splet še v povojih približno 15 do 20 let nazaj, ko je bilo število uporabnikov v primerjavi z današnjim dnem relativno majhno. Strojna oprema je bila občutno manj zmogljiva, same aplikacije pa so bile tudi precej manj kompleksne. Monolitna arhitektura je služila predvsem pri razvoju internih poslovnih aplikacij, vendar se je kasneje, ko se je začela rast uporabniških aplikacij, opisan način uporabil tudi pri slednjih. Večina razvijalcev je omenjeni način razvoja dobro poznala, prav tako pa je bil za takratne razmere in zahteve popolnoma ustrezen.

Z zrelostjo spleta in pojavom računalništva v oblaku, podrobneje IaaS (Infrastructure as a Service), PaaS (Platform as a Service) in SaaS (Software as a Service) platform, se je v zadnjih petih letih stanje močno spremenilo. Velika podjetja vseh področij z veliko količino prometa so spoznala, da omenjena arhitektura ne ustreza njihovim željam in potrebam. Če povzamemo zgornje točke, potrebujejo visoko zmogljive aplikacije, ki jih lahko enostavno in efektivno posodabljajo, da ustrezajo njihovim hitro spreminjajočim poslovnim modelom. Monolitna arhitektura za moderne, velike aplikacije ne pride več v poštev, saj današnjim realnim potrebam ne zadosti v dovoljšnji meri.

V ta namen se je v zadnjih letih razvila arhitektura mikrostoritev kot alternativa tradicionalni monolitni. Mikrostoritve olajšajo razvoj današnjih velikih modernih aplikacij in naslovijo tako pomanjkljivosti kot težave, ki jih srečamo pri monolitnih aplikacijah.

V nadaljevanju predstavimo podrobnosti arhitekture mikrostoritev in povemo, kako se te razlikujejo od tradicionale monolitne arhitekture. Razdelali bomo tudi, kako se mikrostoritve razlikujejo od SOA ter njihove prednosti in slabosti.

2.1 Opis mikrostoritev

Mikrostoritve predstavljajo arhitekturni pristop za razvoj aplikacij kot skupek manjših storitev, kjer vsaka od njih teče kot samostojen proces in komunicira preko preprostih protokolov, najpogosteje je to REST (Representational State Transfer). Storitve so zasnovane glede na poslovne potrebe oz. domene in so samostojno ter neodvisno nameščene s pomočjo raznih orodij in oblačne infrastrukture. V večini primerov je režija celotne aplikacije minimalna, saj vsako mikrostoritev upravljamo posebej. Razvite so lahko s pomočjo različnega jezika oz. tehnologije in uporabljajo različne podatkovne baze.

Glavna značilnost mikrostoritev, ki vedno velja za vse različice, je dekompozicija neke storitve oz. aplikacije na več manjših obvladljivih (mikro) storitev oz. komponent. Ideja o sestavljanju aplikacij z uporabo različnih manjših komponent je sicer prisotna že dolga leta, npr. uporaba SOA, vendar mikrostoritve naredijo še korak dlje. Komponento namreč definirajo kot enoto oz. skupek programske opreme, ki predstavlja neodvisno zaključeno celoto za opravljanje točno določene funkcionalnosti. Ni samo majhen interni del neke večje aplikacije in s tem še vedno odvisna od razvojnega cikla starševske aplikacije, pač pa popolnoma samostojna storitev s svojim razvojnim in življenjskim ciklom. Povezljivost z drugimi komponentami se doseže z uporabo predefiniranih vmesnikov. Največkrat gre za REST storitve, v nekaterih primerih tudi SOAP (Simple Object Access Protocol) storitve. Vsaka komponenta je samostojno nadgradljiva in zamenljiva, tako da v primeru nadgradnje oz. servisiranja ne vpliva na druge dele aplikacije. Na ta način dosežemo veliko boljšo obvladljivost vseh komponent aplikacije [3][4].

Glavna korist strogo ločenih komponent je neodvisna in samostojna namestitev. V primeru, da je aplikacija sestavljena iz več komponent, bi ob morebitni spremembi ene, bilo potrebno ponovno namestiti vse. Če uporabljamo mikrostoritve, lahko pričakujemo, da bo posodobitev ene komponente povzročila ponovno namestitev izključno te komponente. Druge bodo ostale nespremenjene. Na ta način se izognemo dodatnemu delu in testiranju, prav tako pa dosežemo obvladljivejši razvoj, potek namestitev in manj možnosti napak. Omogoča nam hitrejše in natančno odkrivanje izvorov morebitnih napak in posledično zmanjša čas od odkritja do odprave napak. Seveda popolne izolacije ne moremo doseči, saj komponente ravno tako komunicirajo med seboj, zato je ob morebitni večji spremembi delovanja oz. vmesnika še vedno potrebno te spremembe skoordinirati z ostalimi spremembami in jih po potrebi tudi posodobiti. Kljub temu je cilj dobre arhitekture mikrostoritev minimizirati takšne primere s pomočjo točno določenih meja (zaključenimi funkcionalnostmi) med komponentami in ustreznim verzioniranjem ter določanjem vmesnikov komponent.

Z dekompozicijo aplikacije na več manjših obvladljivih komponent se nam poveča oddaljena komunikacija, ki poteka med komponentami. Kako ta komunikacija poteka, ni predpisano. Pomembno je le, da je konsistentno in točno definirana. V praksi komunikacija najpogosteje poteka preko REST storitev, lahko npr. uporabimo tudi SOAP, EJB, RMI, itd. Ker REST storitve nimajo mehanizma za definicijo vmesnika[1], kot ga ima npr. SOAP z WSDL-jem (Web Service Definition Language), ga je potrebno zastaviti samostojno. Načinov je več, ključno je, da so pravila, ki jih zastavimo, konsistentna in točno definirana. Dobra praksa je, da postavimo pravila, na kakšen način ustrezno dvigujemo verzije glede na implementirane spremembe, tako da lahko ocenimo obseg sprememb. Pravila definiramo s pomočjo specifikacije semantičnega verzioniranja[2]. Verzijo sestavljajo tri številke; prva številka

[1] Obstajajo razni poskusi definicije vmesnika, vendar se v praksi noben ni posebej prijel. Najbolj popularna formata sta WADL (Web Application Description Language) in Swagger (http://swagger.io).

[2] http://semver.org

predstavlja glavno verzijo in nakazuje spremembe v kompatibilnosti za nazaj, druga številka predstavlja manjšo verzijo in nakazuje nove funkcionalnosti, tretja številka predstavlja popravke hroščev. S tem formatom dosežemo visoko stopnjo zaupljivosti pri posodobitvah komponent in lahko že s pogledom na verzijo vemo, ali bodo potrebni popravki na drugih komponentah. Prav tako je dobra praksa, da se določi tudi okvirni format REST klicev, npr. kako bodo izpostavljene relacije, kakšen bo format poizvedb, sortiranje, itd. Definicija bolj podrobnega formata vmesnika lahko občutno izboljša kvaliteto in hitrost razvoja aplikacije.

Pomembno je omeniti, kako velika je pravzaprav lahko storitev, da jo klasificiramo kot mikrostoritev. Njena fizična velikost (npr. velikost repozitorija ali število vrstic kode) za opredelitev ni pomembna, pomembno je le, da neka mikrostoritev opravlja izključno eno zaključeno funkcionalnost. V primeru, da pri razvoju razširjamo funkcionalnosti, jih je potrebno identificirati in nesorodne funkcionalnosti odcepiti v drugo mikrostoritev. Kljub temu je zaželeno, da je ekipa, ki razvija mikrostoritev, relativno majhna, približno 3 do 6 razvijalcev. Programska koda tako ostane relativno obvladljiva za vse člane in omogoča produktivnejše okolje.

Uporaba mikrostoritev prinese tudi določene slabosti. Ker smo aplikacijo razbili na več delov, se je občutno povečal oddaljeni oz. medmrežni promet, ki poteka med storitvami. Taki klici so namreč dražji ("overhead" protokola in latenca omrežja) kot tisti, ki bi se izvedli interno v aplikaciji oz. strežniku. V primeru, da je potrebno spremeniti razdelitev funkcionalnosti (trenutna se je izkazala za neustrezno), nam to lahko povzroči preglavice, saj je potrebno usklajevati popolnoma ločene komponente in razvojne ekipe.

2.2 Organizacija po poslovnih zmogljivostih

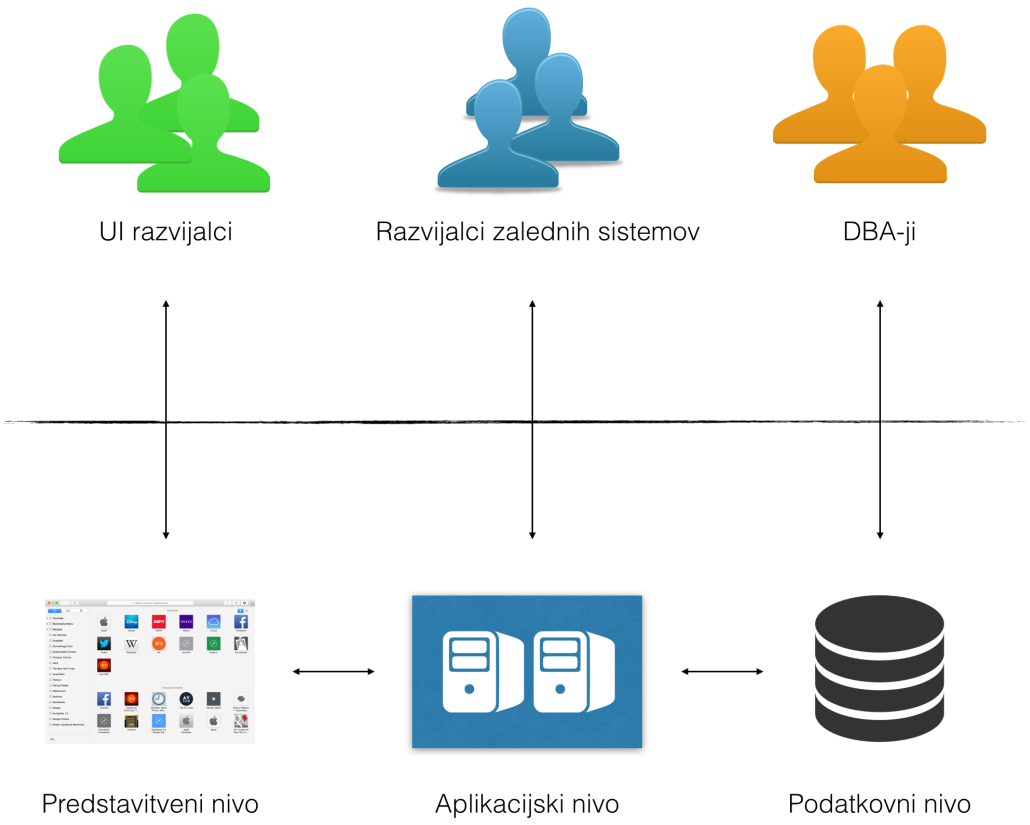
Kadar je potrebno razvoj aplikacij razdeliti med več ekip, lahko to storimo na dva načina, in sicer silosno in domensko. Silosna delitev pomeni, da razdelimo ekipe glede na arhitekturni nivo aplikacije oz. tipe razvijalcev (npr. podat-

kovna baza in uporabniški vmesnik), medtem ko domenska delitev pomeni, da razdelimo ekipe glede poslovne zahteve, ki vsebujejo vse tipe razvijalcev, vendar pokrivajo manjšo funkcijsko področje.

Ko postane aplikacija prevelika za eno samo razvojo ekipo, je standardna praksa, da jo začnemo razbijati na več delov in več razvojnih ekip. Vodstvo se pogosto odloči za delitev aplikacije in ekip glede na nivo tehnologij. Rezultat so ekipe, ki skrbijo izključno za podatkovne baze, za poslovno logiko na strežniku, za uporabniški vmesnik... Kadar so ekipe razdeljene na takšen način, lahko že majhne spremembe poslovnih zahtev oz. modela povzročijo veliko dela in koordinacije, kar stane ekipe veliko časa, ki bi ga lahko porabile za bolj produktivne naloge. Ekipe zato pogosto prisilno implementirajo spremembe v aplikaciji, do katere imajo dostop, da se izognejo temu postopku. Tako sicer pridobijo porabljeni čas, vendar takšno početje čez čas privede do napačno strukturiranih in neobvladljivih aplikacij, kar je lahko na dolgi rok stroškovno neučinkovito. Takšen pristop opisuje Conwayov zakon [5], ki pravi, da bodo organizacije, ki oblikujejo sisteme (v splošnem smislu), pripravile zasnovo, katere struktura je kopija komunikacijske strukture organizacije. V našem primeru to pomeni, da bodo nastale ekipe ustvarile silosne aplikacije glede na arhitekturni nivo. Kvaliteta povezav oz. integracij teh sistemov, kar je pri taki zgradbi ključno, izraža kvaliteto odnosa med razvojnimi ekipami. V primeru, da ta ni najboljša, se to izraža v občutno slabši kvaliteti celotnega sistema.

Zgornji del slike 2.1 prikazuje primer opisane delitve ekip glede na nivo arhitekture aplikacije, ki privede do silosnih aplikacijskih arhitektur in je prikazano na spodnjem delu slike 2.1.

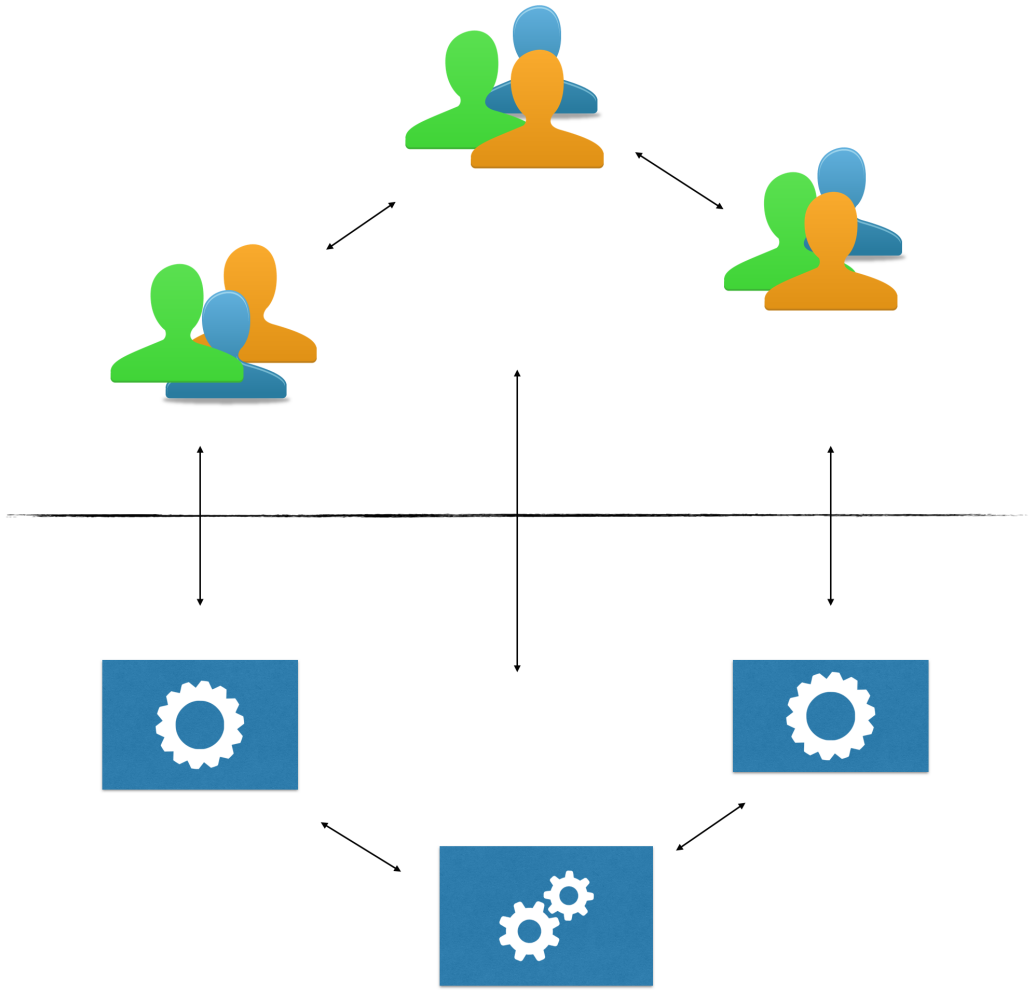
Pristop deljenja ekip je pri arhitekturi mikrostoritev drugačen. Namesto da jih delimo glede na tehnološki oz. arhitekturni nivo, to storimo glede na poslovne zahtevnosti in zmožnosti, kar prikazuje slika 2.2. Tako dobimo storitve, ki vsebujejo vse sloje arhitekture in rešujejo svoje poslovno področje oz. domeno. Vsebujejo uporabniški vmesnik, poslovno logiko v zaledju kot tudi podatkovne storitve. Zagotoviti moramo, da enega poslovnega področja



Slika 2.1: Prikaz silosne delitve ekip in arhitekture (Conwayov zakon)

oz. domene ne pokriva več storitev in se tako med seboj ne prekrivajo glede funkcionalnosti. Posledično so tudi sestavljene ekipe večdimenzijske in vsebujejo vse potrebne kompetence za uspešen razvoj storitev od repa do glave.

Če primerjamo sliko 2.2 s sliko 2.1, slednja prikazuje delitev ekip glede na poslovno domeno in vsebuje razvijalce, ki pokrivajo celoten arhitekturni spekter storitve, kar je prikazano z različnimi barvami razvijalcev - vsaka barva predstavlja en tip razvijalca, kot lahko vidimo na sliki 2.1. Spodnji del slike 2.2 prikazuje rezultat delitve, samostojne (mikro)storitve. Komunikacija med ekipami vsebuje koordinacijo za integracijo medsebojnih storitev, ne pa koordinacije za implementacijo jedra poslovne domene.



Slika 2.2: Prikaz delitve ekip glede na poslovno domeno

Velike monolitne aplikacije lahko vedno razbijemo glede na poslovno področje oz. domeno, vendar v praksi to ni pogosto, saj se velikokrat poslovna logika močno prepleta. Kasneje je potrebno med razvijalci uveljavljati visok nivo discipline. V vsakem primeru je zelo priporočljivo, da se tudi monolitne aplikacije poslužujejo takšne delitve.

2.3 Decentralizirano upravljanje aplikacij

Pomembna posledica monolitne arhitekture in centraliziranega upravljanja aplikacij je nagnjenost k uporabi enakega programskega jezika oz. tehnologije za razvoj vseh delov aplikacije. Tudi pri organizacijah, ki imajo več različnih ali nesorodnih projektov, je pogosto, da se odločijo za uporabo enake tehnologije za vse svoje projekte. Opisani pristop ima svoje prednosti, npr. lažji nadzor nad aplikacijami, večjo pomoč med razvijalci različnih aplikacij, itd. , vendar je praksa pokazala, da je zelo šibak. Ne obstaja programski jezik oz. tehnologija, ki bi bila primerna za vsak aktualen problem. Velikokrat se zgodi, da rešujemo problem s tehnologijo, ki ni namenjena temu, kar privede do slabših aplikacij, večjih razvojnih časov, manj obvladljive kode in več napak, npr. razvoj dinamičnih spletnih aplikacij v brskalniku z uporabo tehnologije JSF (Java Server Faces). Monolitne aplikacije je do neke mere možno razbiti na različne tehnologije, kar ni ravno pogosto oz. ponavadi prinese kup preglavic in težav.

Mikrostoritve ne omogočajo samo razvoja različnih komponent v različnih tehnologijah, pravzeprav tega celo spodbujajo. Osnovno dejstvo, da so vse storitve samostojne in neodvisne, privede do tega, da je popolnoma vseeno, katera mikrostoritev se razvija v kateri tehnologiji. Komunikacija med njimi namreč poteka po standariziranih vmesnikih, kot so REST, SOAP ali kar HTTP (Hypertext Transfer Protocol). Zanje velja da so tehnološko agnostične in tako neke poljubne storitve sploh ne zanima, v kateri tehnologiji je narejena druga storitev, s katero želi komunicirati.

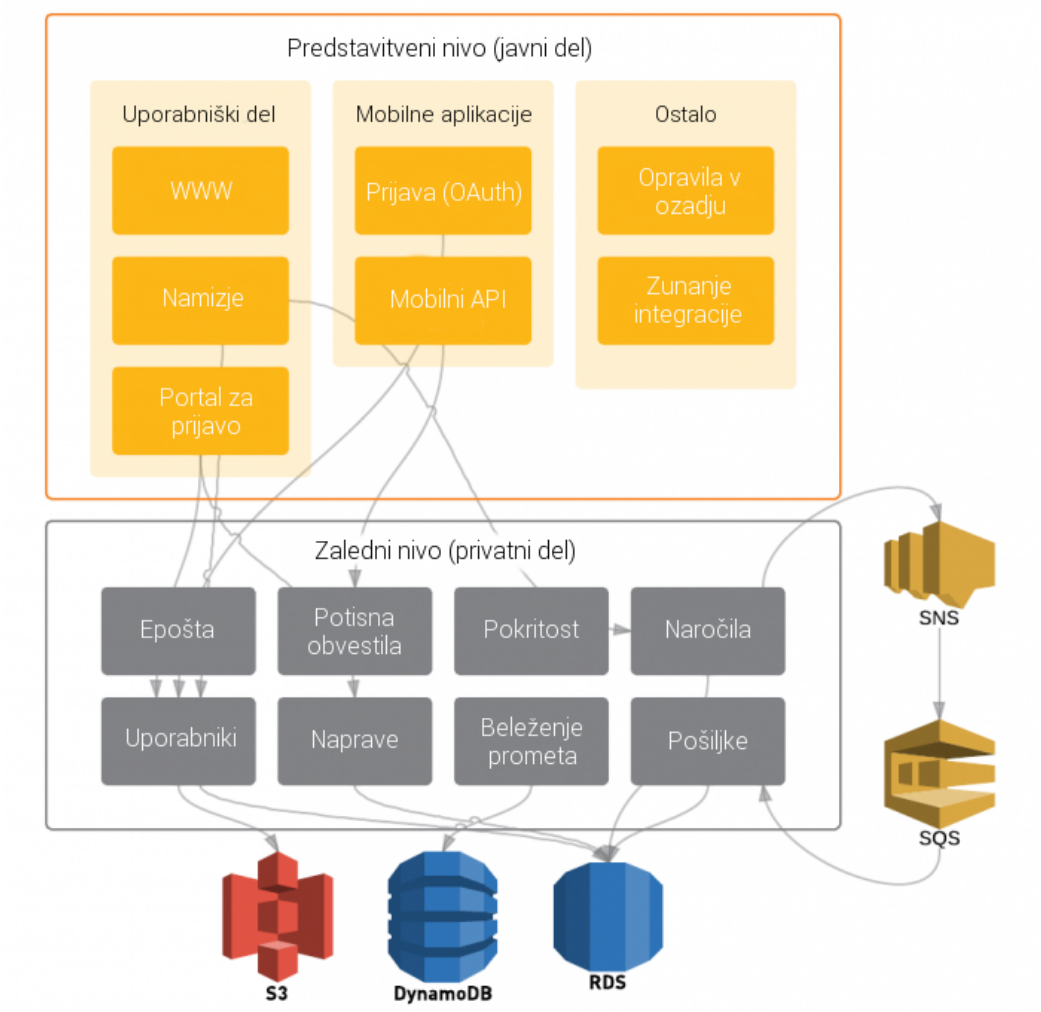
V praksi omenjeno pomeni, da v primeru razvoja SOAP storitve, ki jo bo uporabljala neka vladna aplikacija, uporabimo Javo EE, saj vsebuje eno izmed najboljših orodij za razvoj SOAP storitev. Če npr. potrebujemo visoko zmogljivi HTTP namestnik (ang. proxy), ki usmerja zahteve glede na vnose v bazi, uporabimo ogrodje Node.js, ki ponuja izjemno prepustnost glede na druga ogrodja in visoko zmogljivo bazo Redis, ki hrani vse podatke v pomnilniku. V podobnem primeru, če imamo veliko spreminjajočih dokumentov, ki jih želimo shranjevati, zamenjamo relacijsko podatkovno bazo z NoSQL

(Non SQL) bazo. Takšna fleksibilnost, ki smo jo prikazali v primerih, nam omogoča popolnoma proste roke glede izbire, saj uporaba ene tehnologije na eni storitvi ne vpliva na uporabo drugih tehnologij v drugih storitvah in če v prihodnosti odkrijemo, da izbrana tehnologija ne ustreza našemu problemu, jo lahko enostavno zamenjamo, ne da bi vplivali na ostale komponente. Seveda z izbiro ne smemo pretiravati, saj uporaba druge tehnologije samo zato, ker to lahko naredimo, nima pretiranega smisla. Dejstvo je, da lahko večino problemov rešimo s skrbno izbiro manjšega nabora tehnologij, ki ga nato premišljeno uporabimo za zadani problem.

Ekipe, ki razvijajo mikrostoritve, pogosto odkrijejo, da je velik del zasnove aplikacij enak pri vseh. Prav tako je pogosto, da se ponavlja marsikateri del poslovne logike. V tem primeru se skupne funkcionalnosti oz. zasnove implementirajo v skupne knjižnice, ki se nato interno objavijo za uporabo. Z rastjo odprtokodnih projektov in skupnosti GitHub-a organizacije vedno več orodij objavijo kot odprtokodnih, da lahko dobijo povratne informacije od skupnosti oz. celo sprejmejo popravke ali dodatne funkcionalnosti. Pri monolitnih aplikacijah se lahko skupne zadeve prenesejo v skupne knjižnice, vendar tega ne srečamo ravno pogosto, ker ni veliko potrebe po tem. Pri mikrostoritvah smo v to prisiljeni, saj si ne delijo nobenih skupnih odvisnosti.

Netflix je eno izmed podjetij, ki veliko prispeva v odprtokodno programsko opremo. Veliko njihovih javnih repozitorijev vsebuje orodja, ki so jih interno razvili za potrebe svojih aplikacij [6][7]. Vse skupaj imajo na voljo več kot 50 repozitorijev [8]. Njihova orodja so veliko pripomogla k razvoju sofisticiranih orodij za nadzor velikih gruč sistemov in integracijo ter orkestracijo velikih sistemov storitev.

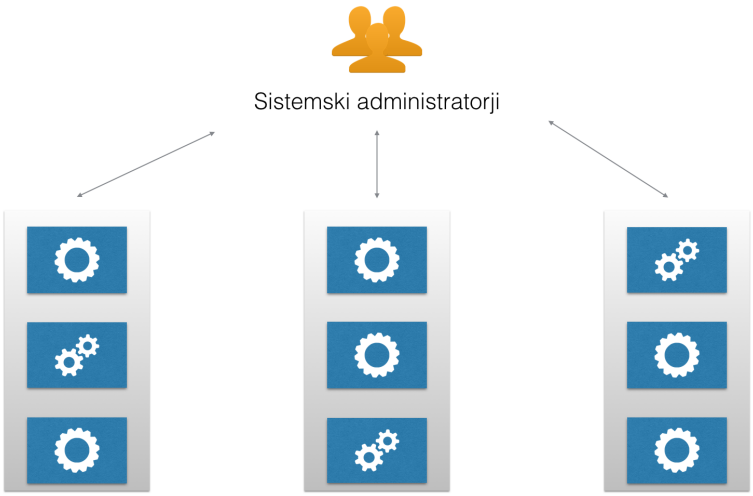
Slika 2.3 prikazuje primer arhitekture mikrostoritev, ki jo podjetje Karma uporablja za svojo aplikacijo v produkciji. V primeru, ki je opisan v [9], vidimo razbitje aplikacije glede na poslovne domene oz. zahteve, uporabo oblačnih storitev Amazonovega oblaka za lažje upravljanje in skaliranje ter uporabo pravilnih orodij za zadani problem. Večina njihovih storitev je narejena s pomočjo programskega jezika Ruby, vendar v primeru, ko Ruby ni



Slika 2.3: Prikaz arhitekture mikrostoritev organizacije Karma

primeren za rešitev, uporabijo druge jezike oz. tehnologije, v tem konktretnem primeru sta to Go in Clojure.

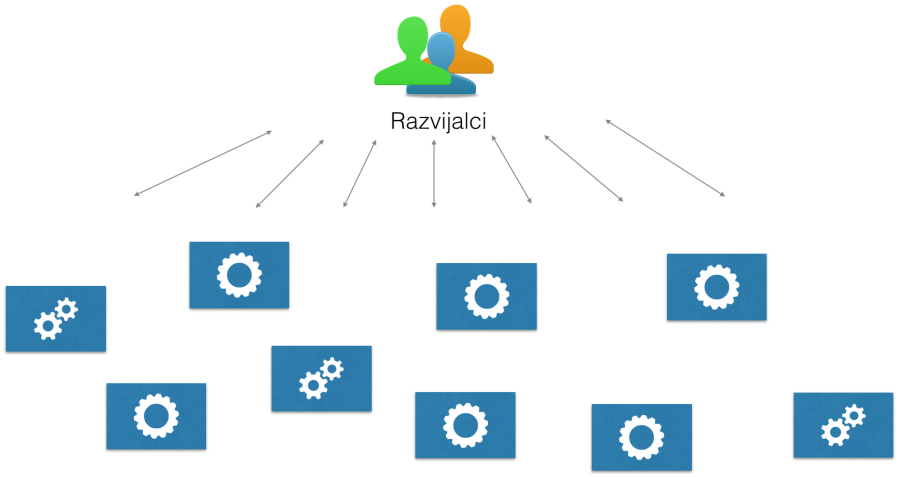
Dodaten razlog za popularizacijo takšnega načina upravljanja je izjemen napredek orodij za avtomatizacijo prevajanja, testiranja in nameščanja aplikacij oz. storitev tako v poljubne zaledne sisteme kot tudi v oblačne infrastrukture (IaaS ali PaaS). Vedno pogosteje je, da ekipe, ki razvijajo neko storitev bodisi samostojno bodisi mikrostoritev kot del večje aplikacije, skrbijo samo za upravljanje, nameščanje in poganjanje svoje storitve. Običajno



Slika 2.4: Prikaz nameščanja več storitev v isti vsebovalnik (aplikacijski strežnik)

naloge padejo na sistemske administratorje in ne na razvijalce. V praksi omenjeno še ni norma, ampak z omenjenim razvojem avtomatizacijskih orodij, je delo vedno lažje in bolj prepuščeno avtomatizaciji, kar pomeni, da naloge lahko opravljajo tudi razvijalci sami. Razvijalci namreč veliko bolje poznajo zgradbo in zahteve aplikacije kot sistemski administratorji, kar pomeni, da lahko samostojno zasnujejo optimalnejši način način namestitve ter se temu že v času razvoja storitve prilagodijo, zato je manj problemov, ko je storitev potrebno dejansko namestiti. Tudi takrat, ko pride do napak, jih lahko razvijalci hitreje in učinkoviteje odpravijo. Razvijalci so prisiljeni v pisanje boljše kode, ker morajo probleme reševati sami, ko se kaj zalomi. Spet je Netflix primer organizacije, ki uporablja takšen sistem razvoja in namestitev [10].

Slika 2.4 prikazuje tradicionalni način nameščanja aplikacij v skupne vsebovalnike, ki so prikazani z večjimi sivimi okvirji. Lahko so v obliki aplikacijskega strežnika, ki izvaja več storitev, ali pa kar v obliki operacijskega sistema in vsebujejo več različnih storitev, ki so prikazane z modrimi okvirji. Za takšne sisteme skrbijo sistemski administratorji, ki velikokrat niso najbolje koordinirani z razvijalci. Vsaka storitev je lahko sestavljena iz več



Slika 2.5: Prikaz nameščanja vsake storitve kot samostojen proces

logičnih komponent (npr. pri Javi ločitev poslovne logike z REST vmesniki), kar nakazuje različno število zobnikov v storitvi. Slika 2.5 prikazuje način, kjer razvijalci samostojo skrbijo za namestitve svojih storitev. Vse storitve so neodvisne in nameščene kot popolnoma samostojen proces. Opisanega načina se najpogosteje poslužujemo, če uporabljamo oblak, kjer izkoristimo PaaS infrastrukturo. V primeru, da nimamo takšne infrastrukture, lahko namestitev za razvijalce postane prekomplicirana, zato delo pogosto opravijo sistemski administratorji.

Z naštetimi načini se oddaljimo od tradicionalnega modela upravljanja (ang. governance) in omogočimo agilnejši in samostojneši model upravljanja, v katerem imajo razvijalci ekip večjo svobodo glede celotnega življenskega cikla storitve. Rezultat je višja kvaliteta storitev in večja motivacija razvijalcev.

2.4 Decentralizacija upravljanja podatkovnih modelov in baz

Poleg ločitve aplikacije na več delov je potrebno omeniti tudi ločitev odgovornosti, vsebin in strukture podatkov v podatkovnih bazah vsake mikrostoritve.

Pri manjših organizacijah večinoma ne predstavlja nobenega problema, saj je podatkovni model, ki ga uporabljajo, dokaj majhen. V tem primeru pogosto ne potrebujemo veliko truda, da uskladimo model z vsemi zahtevami. Problemi se začnejo pojavljati predvsem pri večjih organizacijah; večja kot je, več ima različnih oddelkov in težje se je uskladiti glede podatkovnega modela.

Če gledamo abstraktno, lahko trdimo, da se konceptualni modeli podatkov razlikujejo glede na področje oz. oddelek, kar pomeni, da glede na področje oz. oddelek potrebujemo različne informacije istega tipa podatka. V večjih organizacijah, kjer imajo vsaj nekaj različnih oddelkov, gre za tipičen problem. Vzemimo za primer podatke o poslovnih partnerjih. Oddelek za prodajo bo najverjetneje potreboval drugačne informacije o partnerjih, kot jih bo potreboval oddelek za tehnično podporo. Predpostavljamo, da potrebuje kakšen drug oddelek še kakšne druge informacije. Tako dobimo več pogledov na iste podatke. V primeru, da imajo ti oddelki popolnoma ločene sisteme, se hitro zgodi, da se implementacije podatkovnega modela med seboj kar precej razlikujejo. Najpogostejši primeri so: različni atributi, drugače poimenovani atributi in skupni atributi z različnim semantičnim pomenom. Zgodi se tudi, da poslovni partnerji, kot jih imenuje oddelek za prodajo, ne obstajajo v kakšen drugem oddelku [11].

Dokler sistemi oddelkov delujejo samostojno, do problema sploh ne pridemo, kar je v večini primerov nerealna situacija. Slej ko prej pride do potrebe po integraciji raznih sistemov v organizaciji. Nastanejo velike težave, kajti potrebno je identificirati preslikave podatkov. Preslikava zahteva veliko virov, ker so podatki pogosto domensko specifični. Potrebujemo sodelovanje velikega števila ljudi, da se preslikave uskladijo. Omenjeno velja tako za monolitne aplikacije kot tudi mikrostoritve, vendar je dobro razumeti korelacijo med mejami podatkovnega modela in mejami storitev ter v nadaljevanju posledico skupne podatkovne baze, ki je pogosta pri monolitnih aplikacijah.

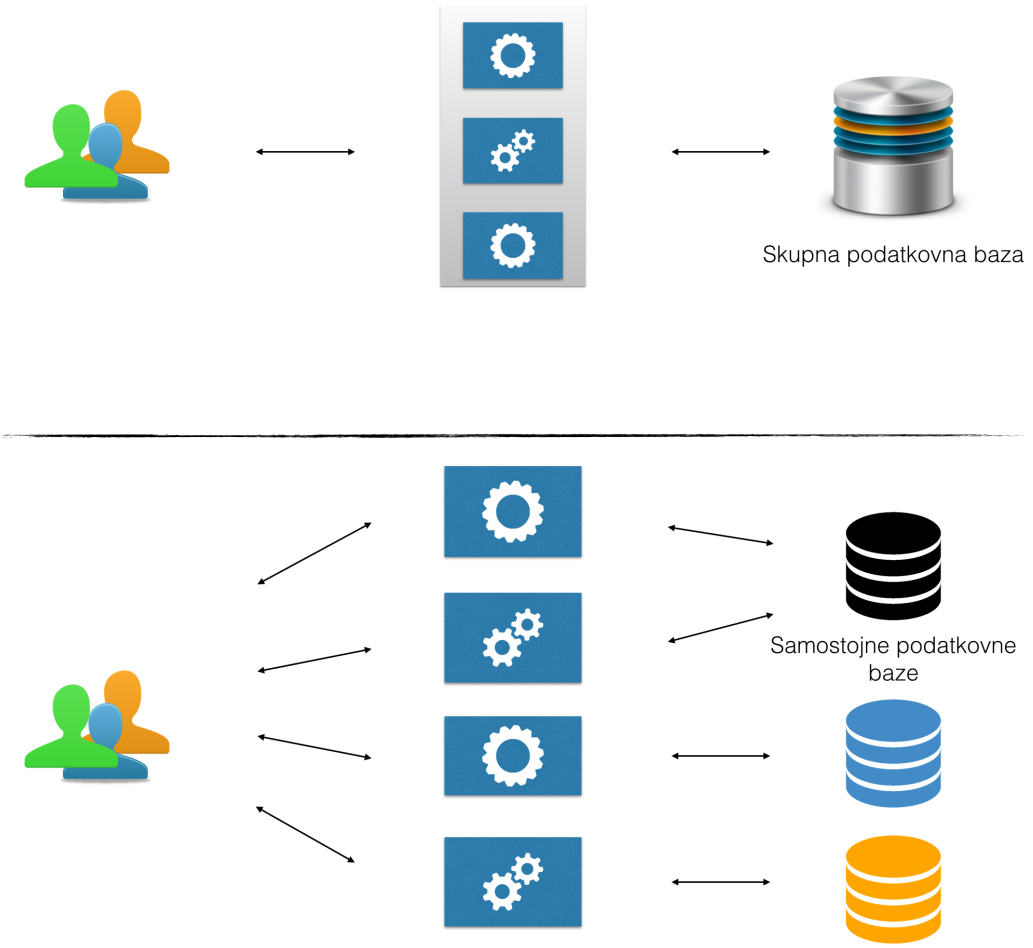
Vrnimo se k mikrostoritvam, ki poleg decentralizacije podatkovnega modela decentralizirajo izbiro glede dejanskega shranjevanja samih podatkov.

Kot je bilo že omenjeno, je pri monolitičnih aplikacijah pogosto, da se uporablja ena podatkovna baza za vse aplikacije, ki jih uporabljajo, kar prikazuje primer, kjer so podatki sicer ločeni, vendar se nahajajo v isti fizični bazi kot tudi primer, kjer se dejansko uporabljajo isti podatki za različne aplikacije. Velike organizacije težijo k uporabi opisanega načina, predvsem zaradi omejitev licenciranja podatkovne baze. Velika večina namreč uporabja komercialne podatkovne baze (Oracle, DB2), ki so najpogosteje v obliki ene logične podatkovne baze (npr. Oracle Exadata). Povedano povzroči, da razvijalci bodisi novih bodisi posodobitve obstoječih aplikacij nimajo nobene izbire glede shrambe podatkov, kar pomeni, da so prisiljeni uporabljati plačljivo podatkovno bazo kljub temu, da morebitni podatki niso primerni za shranjevanje vanjo.

Po drugi strani mikrostoritve silijo k uporabi ločenih podatkovnih baz. Arhitektura predpostavlja, da mora vsaka mikrostoritev za svoje podatke in podatkovno bazo skrbeti sama bodisi kot ločene instance enake baze bodisi popolnoma drugačnega tipa baze (npr. dokumentna shramba, objektna shramba, “in-memory” baza, “key-value” baza, ...). Omenjenemu pristopu z upravljanjem podatkov pravimo “Polyglot Persistence”. Pristop je podrobno opisan v [12], ki v osnovi pravi, da mora aplikacija oz. storitev sama skrbeti za svoje podatke in podatkovno bazo. Povedano vključuje tudi postavitev in posodobitve, za katere se najpogosteje uporabljajo verzionirane migracije. Prav posodobitve se zaradi takšne arhitekture močno poenostavijo. Podatkovna baza vsebuje samo podatke, ki se tičejo mikrostoritve, zato nam ni potrebno skrbeti za druge storitve, ki morebiti uporabljajo del, ki ga želimo posodobiti. Pri monolitnih aplikacijah se, ko želimo vpeljati posodobitve na bazo, pogosto poslužujemo uporabe transakcij, ki vplivajo na večje število storitev. Njihova uporaba sicer močno pomaga pri konsistenci podatkov, vendar prinese določene težave, ko jih želimo uporabiti v okviru več storitev. Distribuirane transakcije je namreč težko robustno implementirati. Javansko okolje ponuja dobro podporo za takšne transakcije s pomočjo sistema JTA (Java Transaction API), vendar so v drugih tehnologijah implementa-

cije distribuiranih transakcij večinoma odsotne. Tudi če odmislimo problem, bomo pri skaliranju storitev hitro naleteli na dodatne težave. Distribuirane transakcije so relativno zamudne in v sistemu z veliko prometa lahko povzročijo veliko čakanja. Podjetje Starbucks je primer podjetja, ki jih zaradi omenjenega razloga ne uporablja [13]. Zanašajo se na alternativo, sočasno skladnost (ang. eventual consistency), ki zagotavlja, da takrat ko nekega podatka ne posodobimo več, vsi dostopi do njega sčasoma vrnejo zadnjo vrednost. Uporaba takšne metode predstavlja nove izzive, vendar se takrat, ko imamo ustrezen način, kako spremembe razveljavimo predvsem pri velikih visoko dostopnih storitvah, pogosto splača žrtvovati integriteto podatkov za hitrejše oz. dodatne odzive. Mikrostoritve se zaradi samega jedra arhitekture nagibajo k uporabi sočasne skladnosti.

Na sliki 2.6 je prikazana razlika med centraliziranim in decentraliziranim podatkovnim modelom oz. bazo. Izbira načina podatkovnega modela je zelo pomembna, kajti v primeru izbire napačnega načina se lahko zgodi, da se izničijo vse izboljšave, ki smo jih pridobili z izbiro arhitekture mikrostoritev. Zamislimo si, da imamo moderno storitev za pogovor preko spleta, ki vsebuje instance v treh predelih sveta. Izbrali smo arhitekturo mikrostoritev in implementirali samostojne ter neodvisne visoko skalabilne mikrostoritve, s katerimi smo dosegli izjemno prepustnosti pri zahtevah za branje, vendar na nivoju podatkovne baze še vedno uporabljamo strogo skladnost (ang. strong consistency). Podatkovna baza je replicirana čez tri predele sveta, zato vsak zapis (sporočilo) zaradi transakcij (zapis se mora potrditi na vseh treh lokacijah) traja več sekund. Kljub odlični skalabilnosti mikrostoritev opisano napravi celotno storitev neuporabno. Pomembno je torej, da pazimo na ozka grla, kar vključuje podatkovni nivo.



Slika 2.6: Prikaz razlike med centraliziranim in decentraliziranim podatkovnim modelom

2.5 Primerjava arhitekture mikrostoritev s tradicionalnimi monolitnimi storitvami

Dodana vrednost arhitekture mikrostoritev je najbolj očitna, ko jo primerjamo z obstoječo monolitsko arhitekturo. Omenili smo že, da se je arhitekturni stil monolitov pojavil v povojih interneta. Danes so aplikacije izjemno popularne tako med manjšimi organizacijami kot tudi med velikimi. Spletne

aplikacije so najpogosteje sestavljene iz treh elementov; podatkovne baze, ki vsebuje vse podatke, ki jih aplikacija želi hraniti (najpogosteje je to relacijska baza npr. Oracle), poslovne logike, ki izvaja potrebne poslovne zahteve in uporabniškega vmesnika, ki se prikaže končnemu uporabniku. Poslovna logika bo sprejela zahtevke, pridobila zahtevane podatke iz podatkovne baze, ustvarila zahtevan HTML (HyperText Markup Language) dokument in ga vrnila uporabniku. Monolitna aplikacija vsebuje vse naštete komponente v eni sami izvajalni enoti ne glede na njeno kompleksnost. Aplikacije postanejo zelo velike in neobvladljive. Programska koda znotraj aplikacije je tudi v veliki večini tesno sklopljena, kar hitro privede do problemov pri izvajanju, saj napaka v enemu delu poruši celotno aplikacijo. Monolitne aplikacije lahko razvijamo modularno, vendar so na koncu še vedno zapakirane kot ena enota.

Ko začnemo z razvojem neke nove aplikacije, je takšen način popolnoma ustrezen. Dokler sta obseg in kompleksnost aplikacije relativno majhna, se slabosti monolitne arhitekture sploh ne pokažejo. Aplikacija teče v enem samem vsebovalniku, v nekaterih primerih celo v enem samem procesu. Zelo primerna je za razvoj, testiranje in namestitev. Tudi skaliranje manjših aplikacij ne predstavlja večjih težav - to lahko storimo s postavitvijo več enakih instanc, med katere nato razporejamo zahtevke. Težave, ki jih poskušajo rešiti mikrostoritve, se začnejo takoj, ko se obseg aplikacije začne večati in sočasno tudi večati promet in ko jo želimo namestiti v oblak. Problemi nastanejo tudi, ko želimo aplikacijo razbiti na več delov bodisi zato, ker želimo delo razdeliti na več ekip bodisi zato, da bi povečali preglednost. Kljub vsemu tudi mikrostoritve niso rešitev vseh problemov, ki se pojavijo med razvojem, vendar predstavljajo pomemben korak k evoluciji arhitektur aplikacij in prisilijo razvijalce k pisanju boljše kode. Monolitne aplikacije imajo še vedno svoj prostor, predvsem zaradi prisotnosti starih aplikacij in aplikacij z manjšim številom uporabnikov, vendar menimo, da bi večini aplikacij koristilo, če bi vsaj delno vpeljali mikrostoritve.

Slika 2.7 prikazuje glavne visokonivojske razlike med obema arhitekturama. Levi del slike 2.7 prikazuje monolitno aplikacijo, ki vsebuje vse zadane

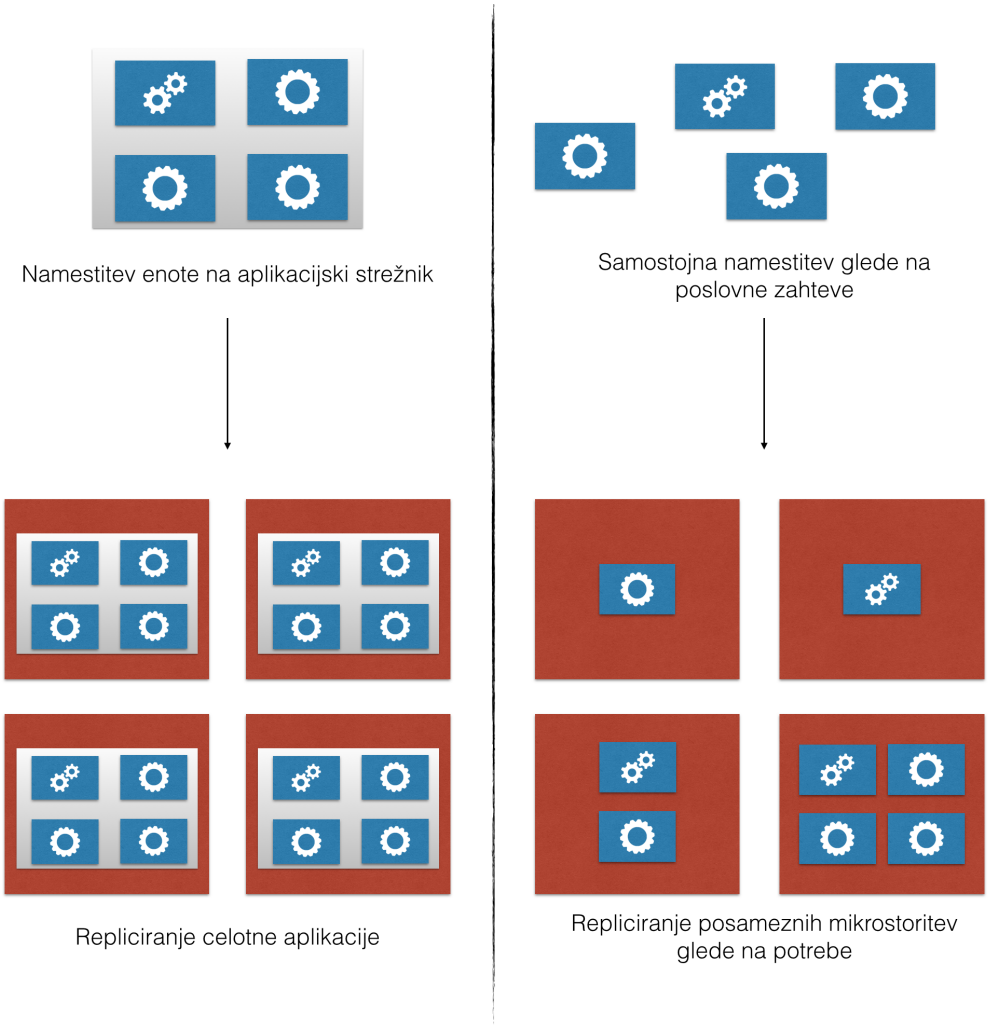
funkcionalnosti končne aplikacije. Zapakirana je v logični vsebovalnik (npr. v Javi WAR/EAR) in nameščena na aplikacijski strežnik, ki skrbi za več aplikacij hrati. Hitro lahko vidimo, da smo takrat, ko želimo posodobiti ali dodati določeno funkcionalnost, primorani ponovno namestiti celotno aplikacijo. Če želimo aplikacijo skalirati, moramo replicirati celoten aplikacijski strežnik in posledično celotno aplikacijo. Desni del slike 2.7 prikazuje mikrostoritve kot samostojne in neodvisne enote, razbite glede na funkcionalnosti, ki skupaj sestavljajo celotno aplikacijo. Ločeno jih zapakiramo in samostojno namestimo. Mikrostoritve med seboj komunicirajo s pomočjo REST storitev. V primeru, ko želimo aplikacijo skalirati, lahko repliciramo želene mikrostoritve individualno, saj pogosto ni potrebno skalirati celotne aplikacije. Posodobitve enostavno izvedemo tako, da zamenjamo samo tisto mikrostoritev, ki se je spremenila.

Sprememba omogoči veliko večjo fleksibilnost pri zasnovi, razvoju in upravljanju aplikacij. Najhitreje in najučinkoviteje opazimo konkretne razlike, ki jih prinese omenjena zasnova, če pogledamo konkretni primer iz prakse.

2.5.1 Primerjava zasnove aplikacije z mikrostoritvami in monolitno arhitekturo

Predstavili bomo funkcionalnosti aplikacije, nato bomo razčlenili prednosti in slabosti monolitne arhitekture ter mikrostoritev, če bi se odločili za eno izmed njih. Smo razvijalci spletnih aplikacij in zadana nam je bila naloga, da implementiramo spletno trgovino za prodajo oblačil. Primer smo nekoliko poenostavili, saj želimo demonstrirati posledice izbire določene arhitekture in ne zasnovati celotnega sistema. Aplikacija bo vsebovala le dve konkretni funkcionalnosti: pregled dobavljivih oblačil in oddajo naročil. Naš cilj bo narediti zasnovo, ki je zadostila naslednjim zahtevam:

- uporabniki lahko brskajo po seznamu oblačil in njihovih podrobnostih.
- Uporabniki lahko za vsako dobavljivo oblačilo oddajo naročilo.

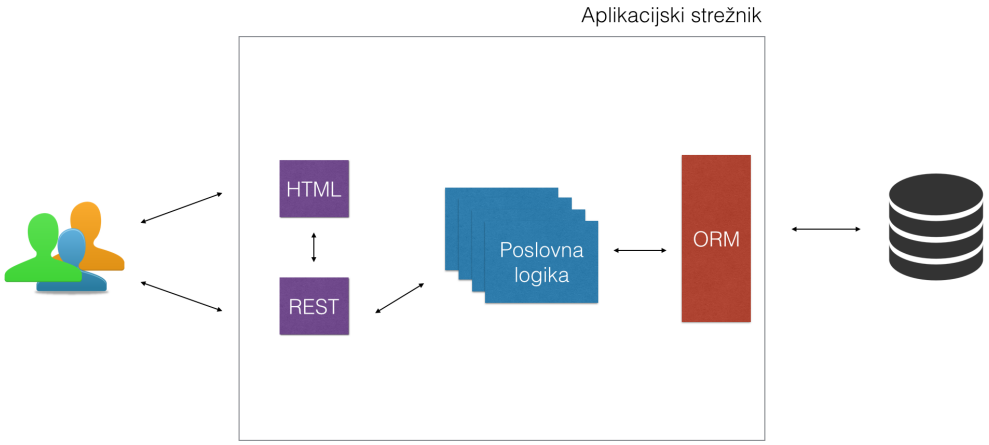


Slika 2.7: Prikaz razlike med monolitno arhitekturo in mikrostoritvami

- Za vsako naročilo je potrebno izvesti plačilo in odposlati artikle.
- Uporabniki lahko pregledujejo svoja naročila.

Sprva bomo aplikacijo zasnovali s pomočjo monolitne arhitekture. V tem primeru bo aplikacija vsebovala logiko za povezavo na bazo (v veliki večini se bo uporabila objektno-relacijska preslikava - ORM) in poslovno logiko, ki bo izvajala dejanske poslovne funkcije. Izpostavljena bo preko REST storitev, preko katerih bo komunicirala HTML5 Javascript aplikacija. Slika

2.8 prikazuje primer tega specifičnega primera.



Slika 2.8: Prikaz monolitne arhitekture primera

Takšna aplikacija bi bila na voljo za namestitev kot skupna enota. Če vzamemo za primer Javo, bi aplikacija vsebovala več javanskih zrn, ki bi komunicirala s pomočjo tehnologije JPA (Java Persistence API) in bila izpostavljena s tehnologijo JAX-RS (Java API for Restful Web Services). Zapakirana bi bila v EAR ali WAR paket vključno s statičnimi HTML5 in Javascript datotekami in nameščeno na aplikacijski strežnik (npr. Oracle WebLogic, JBoss Wildfly, ...).

Identificiramo lahko naslednje dobre lastnosti monolitne arhitekture:

- preprost razvoj - današnji IDE-ji (integrirana delovna okolja, ki omogočajo razvoj aplikacij v izbrani tehnologiji) imajo zelo dobro podporo za razvoj monolitnih aplikacij.
- Preprosto testiranje - aplikacija je v celoti na enem mestu in tako ni potrebno skrbeti za številne integracije oz. koordinacije.
- Preprosta namestitev - vse, kar moramo narediti je, da namestimo aplikacijo in naložimo EAR oz. WAR datoteko na aplikacijski strežnik in skonfiguriramo ustrezne povezave na vire (podatkovna baza, ...).

Ko naša aplikacija postaja popularna in ima vse več obiskovalcev, je potrebno začeti aplikacijo skalirati. Sprva lahko skaliramo aplikacijo s povečavo sistemskih virov oz. nabavo močnejših strežnikov (scale out), vendar je takšna rešitev začasna in kmalu pridemo do meje. Aplikacijo moramo začeti skalirati horizontalno, kar pomeni, da moramo postaviti več vzporednih instanc aplikacije, med katere razdelimo breme, kar odvisno od izbrane tehnologije oz. aplikacijskega strežnika in pogosto ni ravno prijetna izkušnja, podpora oblačnih infrastruktur pa je tudi v večini odsotna.

Zasledimo lahko še naslednje težave:

- velika količina kode pogosto zastraši tako obstoječe kot nove razvijalce. Čez čas postane razumevanje in spreminjanje aplikacije težje, kar povzroči manjšo produktivnost razvijalcev in daljše razvojne čase. Meje med internimi moduli niso točno določene, zato se čez čas premikajo in kasneje tudi propadejo, kar močno zniža kvaliteto kode.

- Preobremenjen IDE - več kot je programske kode v enem projektu, počasnejši je IDE, kar zmanjša produktivnost razvijalcev.

- Preobremenjen aplikacijski strežnik - večja kot je aplikacija, več časa se zaganja, kar močno vpliva na produktivnost razvijalcev, ki veliko časa porabijo, da čakajo nameščanje aplikacije.

- Želeli bi ločiti statične dele predstavitvenega dela aplikacije, kajti zahteve po statičnih datotekah so nepotrebno potratne pri sistemskih virih za poslovno logiko.

- Aplikacije želimo posodabljati po delih. Tako lahko postopoma uvedemo spremembe in se izognemo potencialnim kritičnim napakam. Pri monolitni arhitekturi je možno posodobiti samo celotno aplikacijo. Posodobitev celotne zadeve lahko traja zelo dolgo časa in je ranljiva za napake.

- Ko pride nova verzija uporabljene tehnologije, želimo aplikacijo postopoma posodobiti z novo verzijo.

- Skaliranje je lahko problematično - monolitična aplikacija se lahko skalira samo enodimenzionalno: ali se skalira vse ali pa se ne skalira nič. Kmalu opazimo, da seznam oblačil generira občutno več prometa kot stran z naročili. Želeli bi skalirati samo seznam, ne pa naročil, saj nočemo zapravljati dragocenih sistemskih virov. Prav tako ni mogoče, da bi del aplikacije izvajali v okolju, ki ustreza njegovim potrebam (npr. bolj pomembna je poraba CPU-ja kot pomnilnika), kajti posameznih delov ni možno skalirati neodvisno.

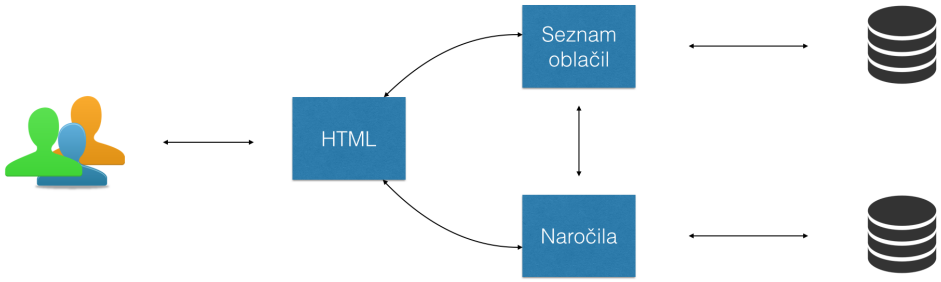
- Identificirali smo del aplikacije, ki ni najbolj primeren za izvajanje v Javi. Za ta del bi želeli uporabiti drugo tehnologijo (npr. NodeJS, Ruby, ...). Monolitna arhitektura se nagiba k uporabi iste tehnologije za celotno aplikacijo. Ne le, da imamo problem pri uporabi druge tehnologije za obstoječi del, tudi pri izbiri tehnologije za prihodnje aplikacije ali komponenete nimamo ravno izbire.

- Če del aplikacije pade, pade celotna aplikacija. V primeru, da se naročila sesujejo, bi želeli, da seznam oblek še vedno funkcionira.

Pri uporabi monolitne arhitekture smo identificirali številne resne probleme, za katere ni enostavne rešitve. Predvidimo, da smo zasnovali aplikacijo s pomočjo arhitekture mikrostoritev. Slika 2.9 prikazuje primer zasnove, v kateri našo aplikacijo razbijemo na več mikrostoritev glede na funkcionalnosti. Vsaka vsebuje logiko za povezavo na bazo, poslovno logiko in izpostavitev s pomočjo REST vmesnikov, preko katerih bodo mikrostoritve komunicirale med seboj. Če ponovno pregledamo seznam problemov, ki smo jih odkrili pri monolitni arhitekturi, lahko vidimo, da imamo enostavne rešitve za vsakega od njih.

- Vsaka mikrostoritev je relativno majhna, kar odpravi večino problemov, ki smo jih identificirali zaradi obsega in kompleksnosti kode.

- Koda vsake mikrostoritve je majhna in enostavno razumljiva. Ker v večini primerov razvijalci delajo samo na eni mikrostoritvi, ostalih delov



Slika 2.9: Prikaz arhitekture mikrostoritev primera

ne vidijo. Tako se razvijalec lažje osredotoči na tisto nalogo, ki jo mora opraviti. Koda med mikrostoritvami je strogo ločena, zato se ne more prepletati, kar dodatno izboljšuje kvaliteto in obvladljivost kode.

- IDE je manj obremenjen, saj je naenkrat odprta maksimalno ena mikrostoritev, ki vsebuje relativno veliko manj kode. Razvijalci so lahko bolj produktivni, saj je IDE veliko bolj odziven.
- Mikrostoritve so po obsegu relativno majhne, zato je potrebno manj inicializacije, kar prinese hitrejše zagonske čase. Omenjeno pride prav predvsem pri skaliranju v oblaku, kjer je nove instance potrebno zagnati v čimkrajšem času.
- Posodobimo lahko vsako mikrostoritev posebej, ne da vplivamo na druge, saj je vsaka mikrostoritev ločen projekt z ločeno kodo in ločeno konfiguracijo. Postopoma uvajamo nove funkcionalnosti in minimiziramo napake.
- Za serviranje naših statičnih delov (npr. HTML5) aplikacije ustvarimo posamezno mikrostoritev, ki je popolnoma ločena od ostalih. Namestimo jo direktno na enega izmed globalnih omrežij za dostavo vsebin

(CDN), ki vsebujejo veliko število strežnikov po celem svetu oz. postavimo najmanjšo možno instanco in jo nato postavimo za CDN.

- Vsaka mikrostoritev je lahko razvita v svojem programskem jeziku ali tehnologiji. Ker mikrostoritve med sabo komunicirajo preko standardnih REST vmesnikov, lahko uporabimo kombinacijo tako kot želimo. Omogoča nam tudi, da zamenjamo mikrostoritev ne da bi spremenili druge, če ugotovimo, da bi bila druga tehnologija bolj primerna. Izognemo se dolgoročni zavezi z določeno tehnologijo, ki se danes zelo hitro spreminjajo.

- Vsaka mikrostoritev je nameščena samostojno in jo lahko vsako skaliramo točno toliko, kolikor potrebuje (npr. glede na obremenjenost). Ni potrebno skalirati vsega in zapravljati sistemskih virov.

- Lahko jih namestimo na praktično oblačno infrastrukturo in skaliramo po želji. V nekaterih primerih je potrebno nastaviti samo število instanc in platforma poskrbi za vse ostalo.

- S pomočjo orodij oblaka lahko vsako mikrostoritev skaliramo posebej dinamično glede na trenutno obremenitev (npr. z uporabo AWS ELB in okolja Heroku).

- Če ena mikrostoritev odpove, druge funkcionirajo normalno naprej. V primeru neka mikrostoritev uporablja nedelujočo mikrostoritev, lahko funkcionalnost začasno izklopimo.

Vidimo lahko, da pri povečanju naše aplikacije in prometa naletimo pri monolitih na veliko težav, za katere imajo mikrostoritve dobro rešitev. Težko je definirati mejo, kdaj je potrebno monolitno aplikacijo razbiti na mikrostoritve. Problem bo velikokrat drugačen od aplikacije do aplikacije, vendar je v veliki večini omenjeno smiselno narediti, ko postane programska koda aplikacije težko obvladljiva (potrebujemo več ljudi ali ekip) in kadar se število uporabnikov občutno poveča (npr. ko aplikacija ni zmožna več ustrezno obdelati zahteve, njeno skaliranje pa je kompleksno). Poudariti je treba, da tako

kot vsaka arhitektura tudi ta ni brez svojih slabosti v primerjavi z monolitno arhitekturo, ki jo podajamo v nadaljevanju.

- Razvijalci morajo nasloviti težave postavitve distribuiranega sistema:
  - potrebno je implementirati komunikacijo med storitvami.
  - IDE-ji so nagnjeni k razvoju monolitnih aplikacij (omenjeno je norma) in velikokrat ne ponujajo dobrih orodij za upravljanje z distribuiranimi sistemi.
  - Obravnavanje primerov uporabe, ki vključujejo več mikrostoritev hkrati, je lahko brez distribuiranih transakcij težko, vendar v večini primerov nujno.
  - V določenih primerih je potrebno veliko koordinacije med razvijalci različnih mikrostoritev, kar lahko predstavlja težave, sploh če ekipi uporabljata drugačno metodologijo.

- Večja kompleksnost namestitve vseh mikrostoritev in konfiguracija korektne komunikacije.

- Težji nadzor celotnega sistema, saj so velikokrat mikrostoritve različnega tipa.

- Večja poraba pomnilnika. Vsaka storitev teče v svojem procesu, kar pomeni, da ima vsaka storitev dodaten "overhead" tehnologije, ki jo uporablja (npr. ena instanca JVM-ja (Java Virtual Machine) pri monolitih proti več instancam enakega JVM-ja). Še posebej se pozna, če je vsaka instanca v svojem operacijskem sistemu, kjer imamo še "overhead" samega operacijskega sistema. Omenjeno danes ni več velik problem, saj veliko storitev teče v Docker vsebovalnikih, ki popolnoma izolirajo procese, vendar ne prinesejo skoraj nobenega dodatnega "overhead-a".

2.6 Primerjava s SOA

Če podrobno pogledamo glavne značilnosti, vidimo, da obstaja veliko vzporednic s storitveno usmerjeno arhitekturo (SOA). SOA je namreč aktualna in v razširjeni uporabi že desetletje. Pravzprav je podobnosti toliko, da se lahko vprašamo, zakaj smo sploh definirali novo arhitekturo, saj na prvi pogled izgleda, da sta arhitekturi bolj ali manj enaki, vendar ob poglobitvi v podrobnosti vidimo, da temu ni tako.

Če povzamemo definicijo storitveno usmerjene arhitekture, je SOA arhitekturni vzorec, v katerem različne komponente aplikacije ponujajo funkcionalnosti preko komunikacijskega protokola [14]. Če jo primerjamo z mikrostoritvami, tudi pri njih aplikacijo razbijemo na funkcionalno zaključene enote, ki med seboj komunicirajo preko komunikacijskega protokola. Kljub temu arhitekturi rešujeta različne probleme. SOA je bila razvita kot strateška iniciativa, katere namen je bil izboljšava vseh poslovnih aplikacij velikih organizacij, tako da se razbijejo na več funkcionalno zaključenih enot in omogočijo večjo fleksibilnost celotne organizacije. Po drugi strani so mikrostoritve nastale zaradi potrebe strukturiranja ene aplikacije in ne celotne organizacije kot take [15].

Mikrostoritve so upravljane in nameščene kot samostojne in neodvisne enote, medtem ko so SOA storitve pogosto implementirane in nameščene kot del ene same večje monolitne aplikacije. Tako so mikrostoritve deležne veliko dodatnih fleksibilnosti, o katerih smo govorili v prejšnjem poglavju, medtem ko so SOA storitve še vedno omejene s težavami monolitnih aplikacij.

SOA poleg storitev, ki jih lahko enačimo z mikrostoritvami, vsebuje še dodatne nivoje, ki skrbijo za upravljanje teh storitev, od uporabe storitvenega vodila za orkestracijo manjših storitev za opravljanje večjih operacij do uporabe procesnih strežnikov za izvedbo poslovnih procesov [16]. Velikokrat so višji nivoji zelo kompleksni, zato se jih veliko razvijalcev visoko zmogljivih aplikacij raje izogiba. Nivoji so v arhitekturi mikrostoritev odsotni, saj je bil namen narediti čimbolj preprosto arhitekturo, ki bo preprosta za razumevanje in bo podpirala agilen ter hiter razvoj novih storitev, hkrati pa

omogočila razvijalcem veliko svobode ter fleksibilnosti. Prav tako v večini primerov, tudi pri velikih organizacijah, teh funkcionalnosti niti ne potrebujemo. Če jih vseeno potrebujemo, lahko še vedno poslujemo s SOA in jo uporabljamo skupaj z mikrostoritvami. Glavna pobuda za ustanovitvev mikrostoritev je bila potreba po preprosti, fleksibilni in močni arhitekturi, podobno kot je bil ustanovljen REST kot odgovor na veliko bolj kompleksni SOAP. Zelo se je obrestovalo, saj pri razvoju novih storitev skorajda ne zasledimo več SOAP-a, ampak samo še REST, kar je pripomoglo k večji zmogljivosti trenutnih storitev.

Kako točno se mikrostoritve umestijo v primerjavi s SOA, je odprta debata. Nekateri popolnoma zavračajo frazo SOA, medtem ko drugi uvrščajo mikrostoritve kot eno izmed oblik SOA. Spet tretji pravijo, da so mikrostoritve to, kar bi SOA originalno morala biti. Menimo, da je pomembno le to, da mikrostoritve ne vzamemo kot alternativo storitveni usmerjenosti, ampak kot evolucijo oz. dopolnitev slednje za podporo modernim visoko razpoložljivim aplikacijam, ki se nahajajo v oblaku. Nekaj večjih organizacij se je že odločilo za vpeljavo mikrostoritev namesto monolitov (SOA), kar nakazuje pozitiven trend uporabe mikrostoritev predvsem pri manjših organizacijah in novih aplikacijah. V prihodnosti pričakujemo vedno večjo uporabo mikrostoritev v večjih organizacijah, še posebej takrat, ko bodo te organizacije začele migrirati svoje produkte v oblak.

Poglavje 3

Metoda za razvoj mikrostoritev v Javi

V poglavju 2 smo podrobno pregledali, kaj so mikrostoritve, kako se primerjajo z obstoječimi arhitekturami in kaj so njihove slabosti ter prednosti. Dobili smo tudi visokonivojski pregled njihovih značilnosti ter obravnavali koncepte, ki jih mora aplikacija vsebovati, da jo štejemo k mikrostoritvam. Potrebno je pregledati, kako lahko koncepte apliciramo na obstoječe programske jezike in tehnologije ter prilagodimo njihov način izvajanja produkcijskemu okolju, da bo skladen z arhitekturo.

Trenutno je za razvoj spletnih aplikacij v uporabi ogromno število različnih programskih jezikov od dinamičnih jezikov, kot so Ruby, Python, PHP, JavaScript in ASP do statičnih jezikov, kot so Java, .NETa, Go, Haskell in preostali jeziki osnovani na JVM (Scala, Groovy, ...). Tudi na nivoju programskih jezikov zasledimo več različnih tehnologij. Na primer Django in Pyramid za Python, Rails in Sinatra za Ruby, Spring, Play in EE za Javo. Nekatere tehnologije bolje podpirajo arhitekturo mikrostoritev, nekatere slabše. Vseh možnih kombinacij tehnologij in programskih jezikov je občutno preveč, zato se bomo osredotočili samo na eno, in sicer Javo. Java je danes eden izmed najbolj popularnih jezikov, poleg tega pa je Java EE ena izmed najbolj popularnih tehnologij za razvoj spletnih aplikacij. Izjemno se uporablja v

poslovnih okoljih in velikih organizacijah ter beleži tudi vedno večjo migracijo iz zastarelih sistemov (npr. Cobol in Delphi). Alternativa Javi EE v teh okoljih je uporaba .NET-a, kaj drugega pa pravzaprav niti ni na razpolago.

Java je za nas še posebej zanimiva, saj gre tradicionalno za jezik, v katerem se razvija velike "napihnjene" monolitne aplikacije. Velikokrat je asociirana s počasnimi, neobvladljivimi aplikacijami, ki se v veliki meri uporabljajo še danes. V resnici so to pogosto starejše aplikacije narejene v času, ko so bile zahteve in obremenitve veliko manjše in drugačne (aplikacije so bile namenjene samo za interno uporabo v organizaciji) in jih organizacije oz. javni sektor še ni posodobil. Razlog je predvsem v tem, da aplikacije še vedno ustrezno funkcionirajo, le malo so zastarele, kar je v veliko primerih irelevantno za poslovni proces. Prihod mikrostoritev prinaša popoln kontrast takšnemu razvoju, zato je zanimivo, kako lahko te koncepte apliciramo na jezik, ki je predominantno uporabljen za razvoj monolitov.

Prihod SOA je predstavljal veliko spremembo v načinu razvoja Java aplikacij in prinesel osnovo za evolucijo Java EE specifikacije, ki je danes še vedno "de-facto" standard. Menimo, da bodo mikrostoritve in koncepti, ki jih te prinesejo, prinesle podobno evolucijo jezika in tehnologije. Pojav oblaka in ogromnega števila orodij za avtomatizacijo ter vedno večja potreba po visoko zmogljivih aplikacijah (tudi v večjih organizacijah, ki do zdaj tega mogoče niso potrebovale - danes je dobra prisotnost na spletu takorekoč obvezna) so prinesle velike spremembe v celoten IT sektor. Tehnologija Java EE in programski jezik Java se bosta morala prilagoditi, kajti v nasprotnem primeru se bodo organizacije vedno bolj nagibale k uporabi drugih tehnologij. To je že začelo podjetje eBay, ki v svojo arhitekturo vpeljuje Javascript in Node.js kot alternativo Javi in Javi EE [17]. Omenjeni trend lahko zasledimo predvsem pri današnjih "startup" podjetjih, ki skoraj nobeno ne uporablja Jave EE. V velikih organizacijah ne bodo kar tako zamenjali tehnologije, kar ima lahko velik pomen, ko manjše organizacije postanejo večje. Obzorje se lahko v svetu IT-ja hitro spremeni. Pomembno je, da je poleg kompatibilnosti za starejše aplikacije (nekaj, kar je glavna prednost Jave), potrebno imeti

v mislih tudi prihodnost.

V nadaljevanju bomo predstavili metodo za razvoj mikrostoritev v Javi, njene koncepte ter vzorce [18]. Obravnavali bomo, kako se razlikuje od trenutno aktualnih metod pri razvoju monolitnih aplikacij ter prikazali, da Java konkurira tudi na področju mikrostoritev.

3.1 Koncepti

Osredotočili se bomo na trenutne koncepte, ki so del Jave EE, ki je prisotna v večini poslovnih aplikacij in tudi ena izmed najbolj popularnih. Java EE vsebuje veliko število različnih konceptov, ki jih lahko uporabimo. Tema diplomske naloge ni podrobna analiza Jave EE, zato se bomo osredotočili samo na dele, ki so prisotni v skoraj vseh današnjih aplikacijah in nato pogledali, kako se ti koncepti prevedejo v uporabo mikrostoritev. V industriji najdemo ogromno število razvijalcev Jave EE, zato je smiselno, da koncepte mikrostoritev izpeljemo iz nje in obstoječim razvijalcem ponudimo enostavno migracijsko pot iz monolitne arhitekture k mikrostoritvam.

Java EE vsebuje skupek standardov, ki obravnavajo različne dele poslovne spletne aplikacije. Razvijajo se in so objavljeni kot JSR-ji (ang. Java Specification Request), ki predstavljajo zahteve in procese Java skupnosti za razvoj in odobritev novih tehnologij ter programskih vmesnikov. Služijo kot načrti za programske vmesnike. Razni ponudniki nato razvijejo implementacije, ki so kasneje del aplikacijskih strežnikov. Osredotočili se bomo samo na glavne komponente, ki jih bomo uporabili tudi pri definiciji konceptov mikrostoritev:

- Servlet - osnovna komponenta, ki skrbi za komunikacijo med odjemalcem in strežnikom preko HTTP protokola.
- JSP (Java Server Pages) in JSF - komponenti, ki skrbita za renderiranje predlog v HTML glede na podatke.
- JAX-RS - kompomenta za razvoj REST spletnih storitev.

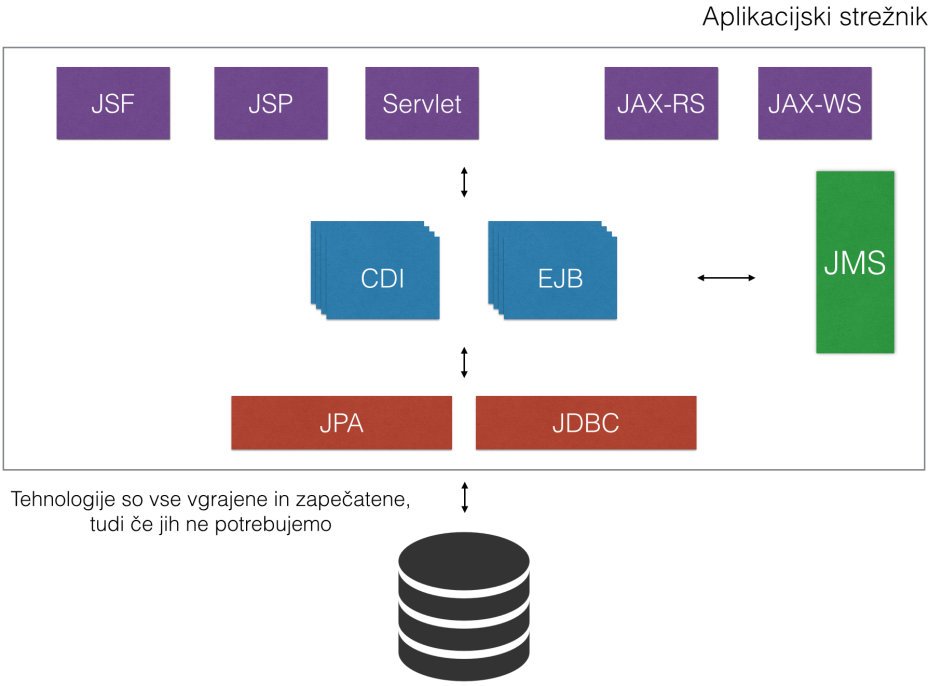
- JAX-WS (Java API for XML Web Services) - komponenta za razvoj SOAP spletnih storitev.
- EJB (Enterprise Java Beans) in CDI (Context and Dependency Injection) - komponenti, ki ponujata DI (Dependency Injection) in podporo za (distribuirane) transakcije. Standarda sta si zelo podobna, EJB izhaja iz starejših verzij, medtem ko je CDI bolj modern standard. V zadnjih verzijah se razvoj nagiba k uporabi CDI komponent. V večini primerov sem umestimo poslovno logiko.
- JPA - komponenta za abstrakcijo in komunikacijo s podatkovno bazo.
- JMS (Java Message service) - komponenta za asinhrono komunikacijo preko vrst (ang. queue) in tem (ang. topic).

Tipična nekoliko bolj kompleksna monolitna aplikacija bo vsebovala vse zgoraj naštete komponente, ki ponujajo rešitve z večino zahtev za spletno aplikacijo. Začnemo z uporabo JPA-ja kot tehnologijO, ki nam ponudi ORM za dostop do dejanske podatkovne baze. Nad tem implementiramo poslovno logiko s pomočjo bodisi EJB bodisi CDI tehnologije. Po potrebi vključimo še asinhrono obdelavo s pomočjo JMS (npr. pošiljanje elektronskih sporočil, da odjemalec ne čaka po nepotrebnem, da se sporočilo pošlje ali pa za daljšo obdelavo podatkov). Nato našo poslovno logiko izpostavimo odjemalcem s katerokoli od naslednjih tehnologij: JSP in JSF (spletna stran), JAX-RS (REST) ali JAX-WS (SOAP). Velikokrat se zgodi, da uporabimo vse tri hkrati (spletno stran za uporabnike, SOAP in REST za druge aplikacije), zato je dobra praksa, da uporabimo CDI oz. EJB nivo za implementacijo poslovne logike, saj jo lahko uporabimo preko vseh treh načinov. Ko se začne aplikacija večati po obsegu funkcionalnosti, jo še vedno vključimo v isto aplikacijo, tako da dodajamo logiko v relevantnem delu oz. tehnologiji. Po izvedbi teh korakov dobimo konceptualno sliko naše aplikacije, ki jo nato namestimo na aplikacijski strežnik. Koncept prikazuje slika 3.1.

Hitro lahko vidimo, da je opisan primer tipična monolitna aplikacija, kjer vse komponente arhitekture ne glede na to koliko funkcionalnosti vsebujejo,

zapakiramo skupaj v paket, ki ga nato namestimo na aplikacijski strežnik, ki upravlja z našo aplikacijo. Vidimo, da imamo konstantno opravka z aplikacijskim strežnikom, katerega uporaba izhaja že iz prvih verzij in se je ohranila do danes.

Pri mikrostoritvah si želimo obdržati vse obstoječe tehnologije, vendar razbiti njihov obseg funkcionalnosti na manjše in bolj obvladljive enote. Povedano pomeni, da bo vsaka mikrostoritev vsebovala tiste komponente, ki jih potrebuje, ampak v manjšem obsegu in pod lastnim nadzorom.



Slika 3.1: Prikaz koncepta monolitne java EE aplikacije

Srce Java EE aplikacije je aplikacijski strežnik, ki deluje kot vsebovalnik, ki skrbi za vire, konfiguracijo in izvajalno okolje aplikacije. Običajno vsebuje vse zunanje povezave (npr. podatkovno bazo, elektronsko pošto, ...) in jih ponudi nameščenim aplikacijam preko standardnega vmesnika JNDI (Java Naming and Directory Interface). Aplikacijske strežnike lahko najdemo v vseh oblikah in velikostih od Tomcat-a, ki vsebuje samo implementacijo Servlet in JSP specifikacije, do Wildfly-a, ki vsebuje implementacijo

celotnega Java EE sklada. Opisan način namestitve je nagnjen k uporabi monolitnih aplikacij, saj zagovarja centralizirano upravljanje vseh aplikacij in njenih povezav. Povedano je v nasprotju z arhitekturo mikrostoritev, kjer vsaka storitev skrbi za svoje vire. Naknadno so pogosto aplikacijski strežniki dokaj obsežni in kompleksni, kar predstavlja dodatno porabo virov in otežuje upravljanje in povečuje efektivno porabo sistemskih virov. Stanje se je precej izbojšalo (Wildfly je občutno "lažji" kot na primer starejši WebSphere), vendar funkcionalnosti aplikacijskega strežnika vseeno ne sodijo v mikrostoritve, zato bi se ga radi popolnoma znebili. Koncept predstavlja probleme tudi pri skaliranju, saj je potrebno vzpostaviti gručo aplikacijskih strežnikov, ki med seboj usklajuje instance aplikacij, pri katerih nimamo želene fleksibilnosti glede skaliranja posameznih aplikacij. Poleg tega danes skoraj noben ponudnik oblačne infrastrukture ne ponuja podpore za kateregakoli izmed aplikacijskih strežnikov, zato se ga pri uporabi mikrostoritev ne poslužimo, ampak skrb izvajanja in povezave na vire prepustimo sami mikrostoritvi.

Odsotnost aplikacijskega strežnika za razvijalce v večini primerov ne bo prinesla hudih preglavic, saj njegovo upravljanje ponavadi ni bila njihova skrb, temveč skrb sistemskega adminstratorja. Za njih je predvsem pomembna uporaba Java EE API-jev oz. tehnologij. Ali se izvaja v aplikacijskem strežniku ali samostojno, v tem kontekstu niti ni pomembno.

Ker smo se znebili aplikacijskih strežnikov, moramo poiskati nov način poganjanja naše kode. Arhitektura mikrostoritev pravi, da mora vsaka mikrostoritev poskrbeti za svoje okolje, kar pomeni, da mora mikrostoritev vsebovati potrebne knjižnice in logiko, da se lahko zažene, saj nimamo neke zunanje programske opreme, ki bi to storila namesto nas. Na trgu trenutno obstaja nekaj rešitev, ki ponujajo integrirano izvajalno okolje. V tem času sta omembe vredni Spring boot[1] in Dropwizard[2]. Obe sta nastali z namenom, da ponudita alternativo aplikacijskim strežnikom in omogočita samostojno in neodvisno namestitev (mikro)storitev. Želeno dosežeta tako, da v aplikacijo

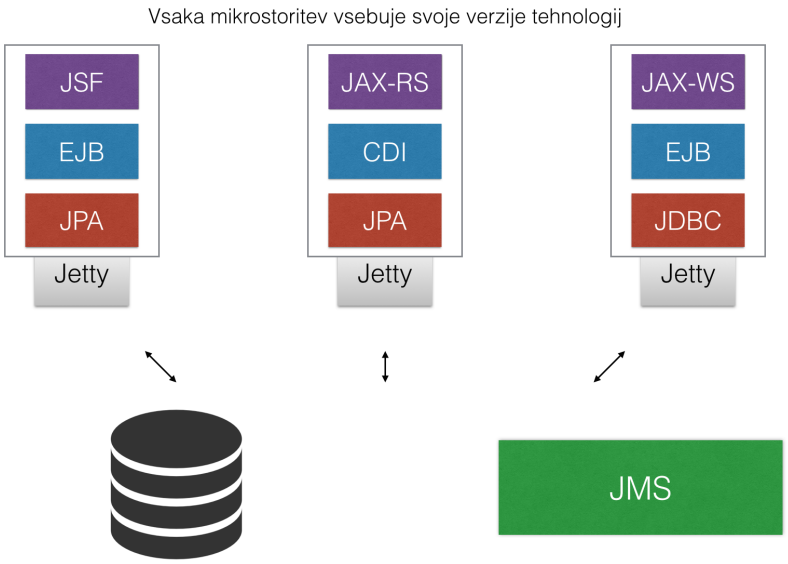
[1] http://projects.spring.io/spring-boot/
[2] http://www.dropwizard.io

vključita vgrajeno različico Jetty-ja ali Tomcat-a, kar bomo uporabili sami, brez njiju. Glavni problem je, da za implementacijo aplikacij ne uporabljata Jave EE. Spring boot uporablja svoje Spring ogrodje, ki se popolnoma razlikuje od Jave EE, Dropwizard pa vsebuje samo dve Java EE tehnologiji, za ostalo pa uporablja različne druge knjižnice. Želimo obdržati vse koncepte in tehnologije, ki jih vsebuje Java EE, saj smo lahko prepričani, da je robustna in je prestala preizkus časa.

Naslednji pomemben koncept je pakiranje aplikacije. Danes je standarden način pakiranja Java spletnih aplikacij uporaba WAR, JAR in EAR pakiranja. WAR je namenjen predvsem pakiranju spletne strani, ki vsebuje servlete, JSP, JSF, JAX-RS in JAX-WS tehnologije. JAR se uporablja predvsem za pakiranje poslovne logike narejene v EJB ali CDI tehnologiji, EAR pa uporabimo za skupno pakiranje več WAR ali JAR projektov skupaj. Na aplikacijski strežnik namestimo bodisi WAR, če gre za manjšo spletno aplikacijo, bodisi EAR, če aplikacija vsebuje kompleksno poslovno logiko in dostop do baze. Nobeden od teh dveh načinov ne podpira samostojnega izvajanja. Končni produkt je pogosto velik monolitni EAR. Namesto takšnega pakiranja raje razbijemo aplikacijo na funkcionalno zaključene enote, opustimo WAR in EAR arhive ter vsako posebej zapakiramo v samostojen in neodvisen JAR. Vseboval bo potrebno kodo za samostojen zagon (npr. uporaba prej omenjenega vgrajenega Jetty ali Tomcat HTTP/Servlet strežnika) in ne bo imel nobenih drugih zunanjih odvisnosti, razen seveda same Jave. Vse, kar bo ta mikrostoritev potrebovala, se bo nahajalo skupaj z njo v končnem JAR-u (vključno s potrebnimi Java EE tehnologijami, ki jih lahko posledično vsaka mikrostoritev izbira zase). Ko želimo aplikacijo pognati, jo lahko poženemo tako, kot bi pognali katerokoli drugo navadno javansko aplikacijo. Koncept mikrostoritev zasnovanih v Javi je prikazan na sliki 3.2.

Če povzamemo, smo definirali naslednje koncepte, s katerimi lahko omogočimo razvoj in izvajanje mikrostoritev z uporabo Jave in Java EE API-jev:

- Uporabimo vse tehnologije, ki jih ponuja Java EE (tako kot pri monolitni arhitekturi), vendar jih uporabimo posamično na nivoju vsake



Slika 3.2: Prikaz koncepta Java EE mikrostoritev

mikrostoritve. Vsaka mikrostoritev lahko sama skrbi za to, kaj in kako uporablja, hkrati pa obseg implementacije ne postane prevelik. Vsaka mikrostoritev mora vsebovati potrebne knjižnice za uporabo Jave EE.

- Izvajalno okolje aplikacije (HTTP in Servlet strežnik) zapakiramo skupaj z aplikacijo, največkrat bo to vgrajen Jetty ali Tomcat, in se ne zanašamo na uporabo aplikacijskega strežnika. Opisani način omogoča mikrostoritvam popolni nadzor nad vsemi aspekti, ki jih potrebujejo za uspešno delovanje (tehnologije, skaliranje, viri, ...), hkrati pa pripomore k zmanjšanju izvajalnega okolja, s tem pa tudi k manjši porabi sistemskih virov.

- Mikrostoritve zapakiramo skupaj z vsemi njihovimi odvisnostmi v samostojni, neodvisni in izvršljivi JAR. Aplikacijo lahko zaženemo povsod, kjer je nameščena Java tako kot vsako drugo javansko aplikacijo, hkrati pa omogočimo lažje skaliranje na raznih PaaS okoljih ter Dockerju.

Našteti koncepti upoštevajo trenutno stanje Jave EE in jo nadgradijo s

podporo razvoja mikrostoritev, ne da bi bilo potrebno spremeniti jedro tehnologije ali pa se poslužiti popolnoma druge, kar je lahko za večjo organizacijo zelo drag in kompleksen proces.

3.2 Vzorci

Java EE vsebuje veliko količino vzorcev (ang. design patterns), ki jih lahko uporabimo. Od načinov uporabe samih komponent Jave EE vse do višjih nivojskih vzorcev, kot so DAO, DTO, fasada, poslovni delegati, šibka sklopljenost, ... Naštete kombinacije tvorijo veliko število dobrih praks pri uporabi Jave EE, ki so podrobno opisane v [19]. Ker so te dobre prakse in vzorci rezultat številnih let razvoja, jih želimo pri razvoju mikrostoritev obdržati vsaj tam, kjer je to smiselno. Glavni cilj mikrostoritev je ponuditi večjo fleksibilnost in neodvisnost, zato lahko vse uveljavljene vzorce brez problema in brez večjih ovir uporabljamo pri migracijah obstoječih aplikacij, a je kljub vsemu pomembno, da se zavedamo, kateri so primernejši za mikrostoritve.

Danes sta v uporabi predvsem dva glavna vzorca razvoja spletnih aplikacij, in sicer: a) pridobivanje podatkov in nato renderiranje HTML datotek na strežniku ter b) izpostavitev surovih podatkov na strežniku preko API-ja (REST ali SOAP) in Javascript statična aplikacija. Pri prvem se predvsem poslužujemo seje s pomočjo piškotkov, ki nam omogočijo prilagojeno izkušnjo glede na uporabnika in stran, na kateri se nahaja. Vse se zgodi na strežniku, klient samo prikaže in morebiti upravlja z animacijami ali čim podobnim. Pri drugem pridobimo podatke preko API-ja in zrenderiramo HTML v brskalniku. API-ji ponavadi ne uporabljajo sej, zanje poskrbijo brskalniki, ki v osnovi držijo stanje (tudi takrat, ko ga zapremo s pomočjo piškotkov in lokalne hrambe).

Pri razvoju mikrostoritev je priporočljiva uporaba vzorca b). Razbitje uporabniškega vmesnika in zaledja pade v koncepte mikrostoritev, saj aplikacijo razbijemo na dve ločeni funkcionalnosti: hrambo in serviranje podatkov ter njihov prikaz. Javascript aplikacijo v resnici sestavljajo samo statične da-

toteke, kar nam omogoči, da zanjo ne porabimo skoraj nič naših sistemskih virov, ampak jo enostavno naložimo na nek globalni CDN, ki poskrbi za redistribucijo datotek po svetu. Renderiranje HTML-ja na strežnik je zelo drago in predstavlja večino porabljenega časa pri zahtevkih za takšne strani, brskalniki pa so v zadnjih letih zelo napredovali, tako da je Javascript za omenjeno delo več kot dovolj usposobljen. Posledica celotnega uporabniškega vmesnika v Javascriptu je, da lahko naredimo veliko bolj dinamično izkušnjo, ker nismo več omejeni na tipičen vzorec zahtevka/odgovora. Aplikacija se izvaja samostojno, podobno kot namizna, le takrat ko potrebuje podatke naredi klic na API, da jih pridobi.

Po drugi strani se močno poenostavi in pohitri zaledje. Ne potrebuje več logike za prikaz podatkov, ampak enostavno vrne zahtevane podatke, kar je občutno hitreje. Najpogosteje so podatki v JSON (JavaScript Object Notation) ali XML (Extensible Markup Language) obliki. Ker ne uporablja sej, se zmanjša poraba pomnilnika, hkrati pa postane vseeno, na katero instanco pride nek zahtevek, kajti določen uporabnik ni vezan na svojo sejo. In ker smo že razvili API za potrebe naše Javascript aplikacije, lahko istega uporabimo tudi za potrebe drugih zunanjih sistemov, kar bi npr. pri opciji a) morali razviti posebej. Opisano omogoča preprostejše dinamično skaliranje aplikacije v oblaku.

Takšna delitev ni dovolj. Če želimo uporabljati vse funkcionalnosti mikrostoritev, kot so opisane v poglavju 2 (skalabilnost, podpora oblakov), mora naša aplikacija slediti zelo pomembnemu konceptu, in sicer aplikacija mora biti "brez stanja" (ang. stateless). Povedano pomeni, da aplikacija oz. storitev pri sebi ne sme hraniti nobenih podatkov, za katere želimo, da so trajni. V praksi našteto pomeni, da mora aplikacija za vse vire, kot so tabelarični podatki, datoteke, slike, sporočila in podobno, uporabiti zunanjo storitev in jih ne sme hraniti pri sebi lokalno, kot je standard večine tehnologij, vključno z Javo EE.

Omenjenega ne smemo mešati z uporabo uporabniške seje. Še vedno jih lahko uporabljamo (v primeru, da uporabljamo npr. JSF), vendar mo-

ramo poskrbeti, da takrat, ko skaliramo aplikacije, zahtevke istega uporabnika pošljemo vedno na isto instanco. Povedano lahko prinese nekonsistentno delovanje (neenakomerno porazdelitev bremena), vendar se zadeve na dolgi rok izravnajo. Temu se lahko izognemo, tako da uporabimo skupno hrambo za seje (npr. zunanjo instanco Redis ključ-vrednost baze), vendar omenjeno prinese dodatno latenco, čemur se pogosto raje izognemo.

Razlog za potrebo po aplikacijah brez stanja leži v tehnikah dinamičnega skaliranja aplikacij v oblaku. Tam se namreč glede na promet ali zahtevano količino postavijo ali uničijo (izbrišejo) instance, ki poganjajo naše aplikacije in med katerimi nato oblak razporeja zahtevke. Tipične so instance tipa Docker ali kar celoten operacijski sistem. Vsaka instanca se lahko izbriše v trenutku; bodisi je ne potrebujemo več bodisi jo oblak premakne nekam drugam. To pomeni, da bi se izbrisalo, karkoli bi shranjevali vanj (poleg aplikacije). Prav tako instance ne vedo ena za drugo, kar pomeni, da neka instanca ne more vedeti za datoteke druge. Rešitev je uporaba zunanjih storitev, npr. osredotočimo se na Amazonov oblak, RDBMS, S3, SQS, SNS, ElastiCache itd. Na omenjene storitve se naše instance aplikacije povežejo, tako da je vseeno kje, kdaj in koliko je instanc.

Če povzamemo, smo definirali dva vzorca, ki jima je potrebno slediti za uspešen razvoj mikrostoritev:

- Aplikacijo razdelimo na zaledni API za serviranje surovih podatkov in Javascript statično aplikacijo za prikaz uporabniškega vmesnika, ki za pridobivanje podatkov uporablja API.
- Aplikacija mora biti brez stanja. Za kakršnekoli trajne podatke moramo uporabiti zunanjo storitev, ki jo uporabljajo vse instance aplikacije.

Obravnavani vzorci za mikrostoritve v Javi EE naredijo aplikacijo bolj preprosto in zmogljivo, a še vedno dopuščajo uporabo vseh vzorcev, ki jih poznamo danes.

Poglavje 4

Implementacija lastnega ogrodja za razvoj mikrostoritev

Razvoj dejanskih mikrostoritev v Javi se sicer v teoriji sliši dokaj enostavno, vendar v resnici naletimo na kar precej težav, ki jih moramo izpostaviti. V poglavju 3 govorimo o konceptih in vzorcih, katerim želimo slediti, medtem ko v poglavju 2 govorimo o dodatnem delu, ki ga moramo opraviti, če želimo razviti uspešne mikrostoritve. V vsakem primeru se razvijalci želijo čim manj ukvarjati s tem, kako bo zadeva dejansko tekla in kaj moramo narediti, da bo vse skupaj funkcioniralo. Prav tako bi se radi čim manj ukvarjali s številnimi konfiguracijami in koordinacijami. Želimo na čim hitrejši način in s čim manj motnjami ter skupne kode (ang. boilerplate) razviti našo aplikacijo.

Potrebujemo ogrodje, ki nam bo omogočalo preprost razvoj mikrostoritev ter nam ne bo v napoto z raznimi podrobnostmi. Hočemo doseči, da obstoječi Java EE razvijalci brez težav začnejo razvijati mikrostoritve, saj bodo tehnologije popolnoma enake. Želimo, da ogrodje podpira ustrezno pakiranje ter konfiguracijo želenih komponent.

Kot smo že omenili v podpoglavju 3.1, sta na trgu aktualni dve ogrodji, in sicer Spring boot in Dropwizard. Obe žal nista primerni za našo situacijo, kljub temu da podpirata razvoj mikrostoritev, saj ne vsebujeta Java EE tehnologij. Spring boot uporablja Spring framework, Dropwizard pa kom-

binacijo določenih odprtokodnih knjižnic. Zasledimo tudi produkt Wildfly swarm, ki pa je še vedno v alfa stanju. Slednji podpira Java EE tehnologije, vendar je v osnovi še vedno Wildfly aplikacijski strežnik, samo da je pripravljen za samostojen zagon poleg aplikacije. Še vedno vsebuje probleme, ki jih prinesejo aplikacijski strežniki ("overhead", skaliranje, konfiguracije, ...).

Nismo zasledili ogrodja, ki bi bilo primerno za izpolnitev vseh naših želja. Ker je tudi to področje še relativno novo in majhno, tudi nismo pričakovali velike podpore s strani obstoječih ogrodij, zato smo se odločili za razvoj lastnega ogrodja, ki bo namenjeno razvoju Java EE mikrostoritev. Ogrodje je odprtokodno, saj želimo pripomoči k popularizaciji in zrelosti Java EE mikrostoritev. Na dolgi rok bomo lahko sodelovali s skupnostjo, ki bo z lahkoto pripomogla kodi in naredila ogrodje še boljše.

4.1 Opis ogrodja

Ogrodje smo razvili sprva kot koncept, s katerim lahko prinesemo arhitekturo mikrostoritev v velik svet Jave EE. Sledi vsem konvencijam, ki smo jih opisali v prejšnjih poglavjih, hkrati pa rešuje glavne probleme, ki jih prinesejo mikrostoritve v primerjavi z monolitičnimi aplikacijami. Koncept smo nato razširili, ga objavili kot odprtokodno orodje in izdali verzijo, ki je primerna za uporabo v produkcijkem okolju. Ogrodje ni zasnovano z namenom, da bi konkuriralo Javi EE (kot je to npr. Spring boot), pač pa je glavno poslanstvo uporaba omenjenih tehnologij na nov moderen, in visoko zmogljiv način, ki bo postavil Javo EE kot primerno platformo za razvoj aplikacij naslednje generacije.

V podpoglavju 2.5 smo omenili tako razlike med mikrostoritvami in monolitnimi aplikacijami kot tudi njihove prednosti in slabosti. Poudarili smo, da je glavna slabost mikrostoritev dodatna kompleksnost pri namestitvi, koordinaciji in konfiguraciji. Prav tako smo v poglavju 3.1 omenili, da se trenutne Java EE aplikacije nameščajo na centraliziran aplikacijski strežnik, tako da nimamo enostavnega načina, kako bi poganjali aplikacijo samostojno in ne-

odvisno. Če hočemo aplikacijo pognati samostojno, jo moramo konfigurirati sami s pomočjo vgrajenih strežnikov (npr. Jetty in Tomcat), ki pa podpirajo samo tehnologiji Servlet in JSP. Če želimo uporabiti katerokoli izmed drugih Java EE tehnologij (teh je namreč precej), jo moramo vključiti in konfigurirati sami, kar pa ni majhen zalogaj. Omenili smo tudi, da moramo aplikacijo zapakirati v ustrezen JAR, ki bo vseboval vse potrebne knjižnice. Dodatno delo je tudi z nameščanjem in vzdrževanjem aplikacije. Zaželeno je, da razvijamo mikrostoritve brez stanja, kar zahteva dodatno konfiguracijo. Vidimo lahko, da se hitro nabere veliko dela, preden začnemo razvijati mikrostoritve. Če želimo uporabiti Javo EE, moramo vse narediti sami, za kar ponavadi nimamo ne dovolj časa ne kompetenc in je že osnova naše aplikacije nagnjena k napakam.

Ogrodje rešuje vse naštete probleme in jih zapakira v enostaven in majhen paket. Ogrodje avtomatizira konfiguracijo in koordinacijo mikrostoritev ter poenostavlja njihove namestitve v razne oblake. Za razvoj uporablja standardne Java EE tehnologije in API-je, kar je ključnega pomena, saj si ne želimo, da bi morali obstoječi razvijalci osvojiti še eno tehnologijo, kar bi morali storiti, če bi želeli uporabljati npr. Spring boot. V današnjem času nasičenega trga novih tehnologij je zelo dobrodošlo, kajti danes skoraj vsako novo moderno ogrodje zahteva od razvijalca, da se nauči veliko zadev na novo. Potemtakem je ogrodje idealno za obstoječe Java EE razvijalce, ki lahko še vedno uporabljajo svoje kompetence in leta izkušenj, a postopoma začnejo prehajati iz razvoja monolitnih aplikacij v razvoj mikrostoritev. Po drugi strani bodo večje organizacije lažje pristale na uporabo ogrodja, saj jim ni potrebno ponovno šolati svojih razvijalcev oz. iskati novih kadrov, kar predstavljal velike stroške in začasno zmanjša produktivnost. Osnovni koncept mikrostoritev omogoča, da se lahko le-te in ogrodje začnejo uvajati postopoma in po potrebi, kar pomeni, da ne predstavljajo velike zaveze k dolgoročni uporabi in s tem minimizirajo tveganje.

Ogrodje vzame številne Java EE komponente in jih z mikrostoritvijo avtomatsko zapakira v samostojen in neodvisen JAR. Poleg tega je ogrodje po-

polnoma modularno. To pomeni, da lahko s pomočjo enega izmed številnih orodij za upravljanje z odvisnostimi in prevajanjem kode, kot so Maven, Gradle, Ivy in podobni, razvijalcu omogočimo, da si posamično izbere, katere komponente Jave EE želi vključiti v svojo aplikacijo. S tem ponudimo še dodatno fleksibilnost razvoja, ki jih druga orodja in aplikacijski strežniki ne morejo ponuditi. Novejše verzije vključujejo t.i. leno nalaganje (ang. lazy loading), kar pomeni, da se aktivitajo takrat, ko so potrebni, vendar so še vedno prisotni v samem izvajalnem okolju. Omenjeno pomeni, da takrat ko razvijalec potrebuje samo JPA in JAX-RS, ni potrebno vključiti preostalih tehnologij, kar naredi mikrostoritev občutno manjšo in bolj učinkovito. Če potrebuje večji sklad, kot je npr. EJB, JAX-RS, JMS, JSF, ga lahko prav tako vključi in izkoristi celoten nabor Java EE tehnologij. Takšna stroga modularnost Java EE komponent nam omogoča še eno dodatno funkcionalnost. Za vsako specifikacijo Java EE komponente obstaja več implementacij (javanska skupnost izda samo specifikacije, po katerih lahko več ponudnikov razvije implementacije le-teh), zato lahko izbiramo, katero implementacijo želimo uporabiti. Npr., če nam knjižnica Hibernate (JPA) povzroča težave, jo lahko zamenjamo za EclipseLink ali pa OpenJPA. Enako velja za ostale kompomenete. To je nekaj, kar pri aplikacijskih strežnikih ni mogoče. Pri njih namreč nismo zaklenjeni le na specifične implementacije, temveč tudi na njihove verzije. Pod vsem tem bosta vgrajena HTTP in Servlet strežnik, ki bosta skrbela za hitro komunikacijo z odjemalci. Naš načrt je, da sčasoma podpremo vse štiri največje produkte (Jetty, Tomcat, Grizzly in Undertow).

V prvi verziji smo se odločili, da bomo implementirali omejen izbor Java EE tehnologij, ker želimo v prvi vrsti doseči stabilnost ogrodja, nato pa dodajati funkcionalnosti. Ogrodje je zasnovano modularno, zato je kasnejše dodajanje tehnologij preprosto. Za vsako implementirano tehnologijo smo se odločili, da bomo uporabili referenčne implementacije, saj te delujejo najbolj korektno glede na specifikacije. V ogrodju so na voljo:

- Servlet 3.1 (Jetty),
- JSP 2.3 (RI JSP),

- EL (Expression Languange) 3.0 (RI UEL),
- CDI 1.2 (RI Weld),
- JPA 2.1 (RI EclipseLink),
- JAX-RS 2.0 (RI Jersey),
- Bean Validation 1.1 (RI Hibernate validator),
- JSON-P (JSON Processing) 1.0 (RI JSONP).

V prihodnosti načrtujemo, da bomo dodali še preostale java EE tehnologije, prav tako pa bomo dodali tudi alternativne implementacije tehnologijam, ki so že na voljo.

Vse izbrane komponente so nato avtomatsko sestavljene z minimalno konfiguracijo razvijalca. Vse nastavitve virov (npr. baza, objektna hramba) so eksplicitno navedene, tako da jih lahko aplikacija prebere iz spremenljivega okolja in tako funkcionira v skoraj vseh PaaS infrastrukturah.

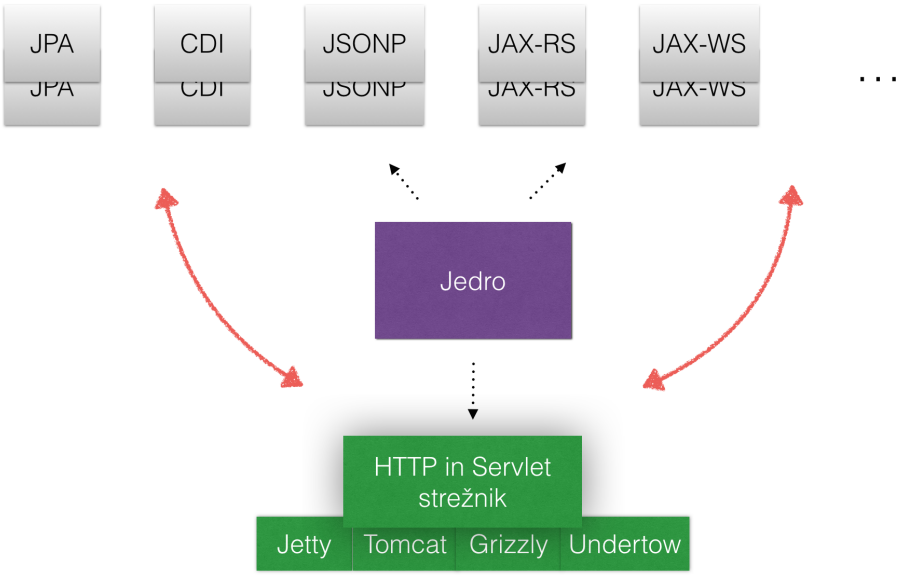
4.2 Zasnova ogrodja

Zasnova ogrodja je popolnoma modularna. Kot smo omenili v poglavju 4.1, je bila ena izmed glavnih želenih funkcionalnosti dinamična izbira Java EE tehnologij. Strukturo ogrodja smo razbili glede na posamične Java EE tehnologije, kar pomeni, da bo vsaka tehnologija imela ena-proti-ena preslikavo z modulom ogrodja. Na ta način bomo dosegli, da lahko z izbiro odvisnosti določimo, katere tehnologije želimo uporabiti, kar bo zagotovilo, da nepotrebnih tehnologij ne bo v izvajalnem okolju, hkrati pa poenostavilo izbiranje. Ker bo vsaka komponenta ločen modul, nam je omogočeno, da lahko številne tehnologije dodajamo postopoma, hkrati pa lahko za eno tehnologijo dodajamo več modulov z različnimi implementacijami, med katerimi bo lahko končni razvijalec nato izbiral.

Pod vsemi moduli oz. tehnologijami potrebujemo HTTP in Servlet strežnik. Ker želimo, da si lahko razvijalec izbere strežnik, ki bi ga rad uporabil, smo te implementirali v ločenih modulih. Razvijalec si bo lahko izbral, na katerem strežniku bodo tekle njegove izbrane tehnologije. Ko združimo vse skupaj, dobimo izjemno fleksibilno ogrodje, s katerim ima razvijalec popolni nadzor.

Vse skupaj povezuje modul jedro, ki bo s pomočjo orodja "ServiceLoader" poiskal vse vključene komponente v izvajalnem okolju, pregledal, če je vse korektno (npr. nimamo podvojenih implementacij tehnologij), jih inicializiral in dodal v izvajanje. Vsaka komponenta mora vsebovati razred, ki implementira ustrezen vmesnik ("EeServer" v primeru, da je komponenta HTTP oz. Servlet strežnik in "Component" v primeru preostalih Java EE komponent), v katerega je implementirana ustrezna logika za inicializacijo komponente. V primeru, da gre za komponento, ki upravlja s HTTP zahtevki, je to najpogosteje preko mehanizma ServletContextInitializer (npr. JAX-RS, JSF, ...), v nasprotnem primeru je potrebno zagnati samo komponento in poskrbeti, da se ustrezno poveže z ostalimi (JPA, Bean Validation, ...). Tako omogočamo, da si lahko vsak razvije dodatne komponente, ki jih vključi. Jedro bo poskrbelo za vse ostale.

Vsak modul se nato zapakira s svojimi odvisnostmi in objavi na centralnem repozitoriju, tako da lahko dostopajo do njega vsi brez dodatnih konfiguracij in repozitorijev. Takšno ogrodje je dostopno tudi razvijalcem, ki uporabljajo kakšno drugo orodje za odvisnosti, npr. Gradle in Ivy. Slika 4.1 prikazuje visokonivojsko zasnovo celotnega ogrodja. Črtkane črte prikazujejo komunikacijo, ki jo jedro izvede med raznimi komponentami za potrebe ustrezne konfiguracije, oranžne črte pa predstavljajo dejansko delovanje aplikacije ob času izvajanja. Spodnji strežnik skrbi za komunikacijo z odjemalci, nato pa zahteve posreduje ustreznim tehnologijam za obdelavo.



Slika 4.1: Zasnova našega ogrodja za razvoj mikrostoritev v Javi EE

4.3 Primer razvoja mikrostoritve

Najboljši način, da demonstriramo način razvoja mikrostoritev s pomočjo našega ogrodja je, da prikažemo razvoj primera kratke mikrostoritve. Vzemimo primer aplikacije, definirane v poglavju 2.5, na osnovi kater razvijamo dve mikrostoritvi, ki bosta skupaj tvorili našo aplikacijo. Prva mikrostoritev naj opravlja nalogo prikaza seznama oblačil, druga pa bo upravljala naročila. v praksi lahko demostriramo arhitekturo mikrostoritev z uporabo ogrodja in Java EE tehnologij.

Za strukturo in postavitev projekta smo uporabili orodje Maven, saj je orodje, v katerem smo razvili samo ogrodje. Če želimo uporabiti kakšno drugo orodje (npr. Gradle, Ivy), lahko brez težav storimo. Ustvarili smo dva projekta, vsak od njiju vsebuje eno mikrostoritev. Zaradi preprostosti in jedrnatosti smo oba projekta ustvarili v istem fizičnem repozitoriju. V realnih projektih bi imeli vsak projekt v svojem repozitoriju, saj si želimo, da je

vsaka mikrostoritev samostojna in neodvisna. Vsaka ima svoje verzioniranje in zgodovino sprememb s pomočjo programske opreme, kot je Git, SVN ipd. Vsaka vsebuje svoje konfiguracije za namestitev v oblaku, ki jo bomo prikazali v poglavju 5.

Odsek 4.1 prikazuje našo zgradbo projektov mikrostoritev, ki jo bomo uporabili v nadaljevanju. Zgradba odraža tipično zgradbo Maven projekta. Vsak modul, vključno s krovnim, vsebuje svojo "pom.xml" Maven konfiguracijsko datoteko. V imeniku "src" vsak modul vsebuje programsko kodo in morebitne teste mikrostoritve. V realnem projektu bi bila imenika kataloga in naročila v svojem repozitoriju. Dodali smo še modul "entitete", ki bo vseboval naše JPA entitete, ki jih bosta mikrostoritvi delili (tako se izognemo nepotrebnemu podvajanju). Predpostavimo, da je zgradba projekta že postavljena in osnovne *pom* datoteke že inicializirane.

```
.
+-- katalog
|   +-- src
|   |   +-- main
|   |   +-- test
|   +-- pom.xml
+-- entitete
|   +-- src
|   |   +-- main
|   |   +-- test
|   |       +-- pom.xml
+-- narocila
|   +-- src
|   |   +-- main
|   |   +-- test
|   +-- pom.xml
+-- pom.xml
```

Odsek 4.1: Zgradba projektov mikrostoritev

Prvi korak je, da vključimo potrebne odvisnosti ogrodja. Omenili smo, da je ogrodje popolnoma modularno, tako da moramo vsako Java EE komponento eksplicitno vključiti. Edina obvezna komponenta je jedro ogrodja. Ogrodje nato avtomatsko zazna, katere komponente so vključene in jih ustrezno konfigurira. Vse komponente, ki so na voljo, so verzionirane skupaj, tako da je priporočljivo, da si definiramo spremenljivko za verzijo ter jo nato uporabimo povsod, kar prikazuje odsek 4.2. Sprva razvijemo mikrostoritev, ki bo prikazovala katalog oblačil, tako da se vsaka koda vnaša v modul "katalog".

```
<properties>
    <ogrodje.version>1.0.0</ogrodje.version>
</properties>
```

Odsek 4.2: Definicija verzije ogrodja (./catalogue/pom.xml)

Nato vključimo jedro, kar prikazuje odsek 4.3.

```
<dependency>
    <groupId>si.fri.ogrodje.ee</groupId>
    <artifactId>ogrodje-ee-core</artifactId>
    <version>${ogrodje.version}</version>
</dependency>
```

Odsek 4.3: Vključitev jedra ogrodja

Jedro samo po sebi ne naredi nič, zato vključimo še vsaj HTTP strežnik, tako da bo lahko ogrodje sprejelo in obdelalo naše zahtevke ter jih posredovalo ustrezni Java EE komponenti. Tokrat izberemo Jetty kot implementacijo HTTP in Servlet strežnik, saj je trenutno edini strežnik, ki smo ga podprli. Omenjeno prikazuje odsek 4.4.

```
<dependency>
    <groupId>si.fri.ogrodje.ee</groupId>
    <artifactId>ogrodje-ee-servlet-jetty</artifactId>
    <version>${ogrodje.version}</version>
</dependency>
```

Odsek 4.4: Vključitev Jetty strežnika

Opisano je minimalni vložek, ki ga moramo izvesti in podpira izvajanje Servlet-ov ter serviranje statičnih datotek. Dodamo še Servlet in HTML datoteko ter mikrostoritev poženemo. Ogrodje tedaj išče vire in konfiguracije v imeniku "webapp", ki se nahaja v imeniku "resources". Omenjeno je edina razlika s klasično Javo EE, kjer je imenik "webapp" poleg imenika "resources" in ne v njem. Deskriptor "web.xml" ni potreben; če ga želimo, ga lahko dodamo in ogrodje ga bo avtomatsko zaznalo ter uporabilo.

Primer HTML datoteke shranjene kot "index.html" v imeniku "webapp" prikazuje odsek 4.5, primer Servleta pa odsek 4.6.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Seznam oblac"il</title>
</head>
<body>
    <p>Tukaj lahko prikazemo seznam oblacil.</p>
</body>
</html>
```

Odsek 4.5: Primer HTML spletne strani

```
@WebServlet("/servlet")
public class SimpleServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request,
                HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/plain");
        response.setStatus(HttpServletResponse.SC_OK);
        response.getWriter()
                .println("Preprost servlet");
    }
}
```

Odsek 4.6: Primer servleta

Jedro ogrodja vsebuje razred "si.fri.ogrodje.ee.EeApplication" z glavno metodo, ki v celoti skonfigurira in požene našo mikrostoritev. Potrebno jo je samo še prevesti in zapakirati. V ta namen uporabimo Maven vtičnik "maven-dependency-plugin", ki bo poleg naše aplikacije vključil še vse odvisnosti. Odsek 4.7 prikazuje konfiguracijo omenjenega vtičnika v *pom* konfiguracijski datoteki. Opazimo lahko, da aplikacija izgleda skoraj enako, kot če bi uporabili samostojno Javo EE.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>2.10</version>
    <executions>
        <execution>
            <id>copy-dependencies</id>
```

```
            <phase>package</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Odsek 4.7: Konfiguracija vtičnika za pakiranje odvisnosti

Nato lahko poženemo Maven pakiranje in z ukazom, ki ga prikazuje odsek 4.8, poženemo našo mikrostoritev.

```
$ java -cp katalog/target/classes:katalog/target\
> /dependency/* si.fri.ogrodje.ee.EeApplication
```

Odsek 4.8: Primer ukaza za zagon mikrostoritve z našim ogrodjem

Obiščemo naslov "http://localhost:8080/" v brskalniku in prikaže se nam datoteka, prikazana v odseku 4.5. Preverimo lahko tudi delovanje servleta, če obiščemo naslov "http://localhost:8080/servlet".

Na podoben način lahko razvijemo tudi mikrostoritev za naročila oblačil. Ta potem vsebuje svoje odvisnosti, kodo in vire, neodvisne od drugih mikrostoritev. Lahko jo poženeno na enak način, kot prikazuje odsek 4.8. Privzeto ogrodje posluša na vratih 8080. Če ga želimo prilagoditi (v primeru, da poganjamo dve mikrostoritvi hkrati), lahko podamo poljuben port s pomočjo spremenljivke okolja "PORT".

Demonstrirali smo, kako lahko začnemo razvijati Java EE aplikacijo z minimalno konfiguracijo ogrodja. Za predstavljeno bi potrebovali veliko časa, če bi funkcionalnosti implementirali sami. Videli smo tudi, kako preprosto je izbrati komponente, ki jih želimo uporabiti. V tem primeru smo uporabili preproste servlete, vendar lahko na popolnoma enak način dodamo še preostale podprte tehnologije. Če bi hoteli uporabiti JPA, CDI in JAX-RS, bi morali dodati še naslednje odvisnosti:

- “ogrodje-ee-jpa”,
- “ogrodje-ee-cdi”,
- “ogrodje-ee-jax-rs”.

Tehnologije lahko normalno uporabljamo, kot bi razvijali aplikacijo za razne aplikacijske strežnike. Vidimo, da ogrodje ne spremeni uporabe močno uveljavljenih tehnologij Jave EE, hkrati pa doda funkcionalnosti, ki jih v slednjih pogrešamo in si jih želimo imeti, še posebej v zvezi z mikrostoritvami.

Poglavje 5

Skaliranje mikrostoritev v oblaku

Uporaba in uveljavitev arhitekture mikrostoritev v spletnih aplikacijah prinese občutno več opcij pri uporabi oblačnih infrastruktur za namestitev, upravljanje in vzdrževanje aplikacije. Zaradi osnovnih lastnosti mikrostoritev, opisanih v poglavju 2, in dejstva, da so mikrostoritve v večini primerov brez stanja (opisano v poglavju 3.2), lahko zanje brez večjih težav uporabimo tako IaaS infranstrukturo kot tudi PaaS. Omenjeno omogoča večjo fleksibilnost pri poganjanju aplikacije in dobro pripravljenost na prihodnost, ko pričakujemo veliko povečanje prometa in števila uporabnikov. Potrebno je zagotoviti, da lahko aplikacijo ustrezno skaliramo z minimalnim balastom in jo izvedemo hitreje glede na trenutni promet in obremenitev. Povedano je potrebno zagotoviti za vsako mikrostoritev posebej, da po nepotrebnem ne zapravljamo sistemskih virov in posledično denarja. Pomembno je, da se zavedamo načinov skaliranja in vemo, katere bomo uporabili pri razvoju in razbitju mikrostoritev, da lahko zanje ustrezno planiramo zasnovo.

V kontekstu mikrostoritev so zanimivi predvsem PaaS oblaki. Ponudijo nam celovito platformo, ki za nas ustrezno upravlja z nizkonivojskimi operacijami (npr. vzpostavljanje virtualnih strojev, namestitev in zagon aplikacije, ustrezna razporeditev zahtevkov, monitoriranje aplikacije, ...), da se lahko

osredotočimo izključno na agilen in hiter razvoj naše aplikacije. Tudi mikrostoritve lahko namestimo v IaaS okolje, vendar moramo v tem primeru vse namestiti in konfigurirati sami, kar se v resnici ne razlikuje veliko od namestitve monolitnih aplikacij. Glavna prednost mikrostoritev je možnost uporabe skoraj vseh PaaS oblakov in vseh njihovih številnih orodij, ki nam močno olajšajo celoten življenjski cikel aplikacije. Če na primer vzamemo monolitne aplikacije v Javi EE, ki jih moramo namestiti na aplikacijski strežnik, lahko ugotovimo, da jih skoraj noben PaaS ponudnik ne podpira. Za namestitev, konfiguracijo in skaliranje aplikacije smo prepuščeni sebi, kar predstavlja veliko dela, saj moramo poleg postavitve virtualnih strojev ustrezno postaviti še več instanc aplikacijskih strežnikov in jih povezati v gručo, v kateri teče aplikacija.

Tudi v primeru, ko ne želimo uporabiti PaaS infrastrukture (npr. v primeru, ko so storitve predrage, ne potrebujemo vseh funkcionalnosti, imamo privaten oblak, ...), lahko mikrostoritve namestimo v poljubno okolje. Omenjeno dejstvo prinese dodatno fleksibilnost glede poslovnih procesov, ki jih pri monolitnih aplikacijah nismo deležni, saj nismo več odvisni od točno določenega okolja in ponudnika, ampak imamo možnost, da aplikacijo prilagodimo okolju, ki nam najbolj ustreza.

V nadaljevanju predstavimo različne načine skaliranja aplikacije v oblaku; kaj predstavlja posamezen način v kontekstu mikrostoritev in kako lahko opisano storimo s pomočjo Jave EE in našega ogrodja ter ali je omenjeno možno storiti z monolitnimi aplikacijami. Pregledali bomo, kaj vsak način pomeni v PaaS in IaaS okoljih ter kako lahko to dosežemo. Za primer PaaS okolja bomo uporabili generično rešitev osnovano na Docker vsebovalnikih, kajti večina današnjih PaaS okolij teče na osnovi Docker-ja . Razlikujejo se predvsem v tem, kakšna orodja imajo za upravljanje teh vsebovalnikov. Prikažemo še primer, kako lahko na enostaven način (par ukazov v terminalu) namestimo ter skaliramo primer mikrostoritve, razvite s pomočjo ogrodja, ki smo ga prikazali v poglavju 4.3.

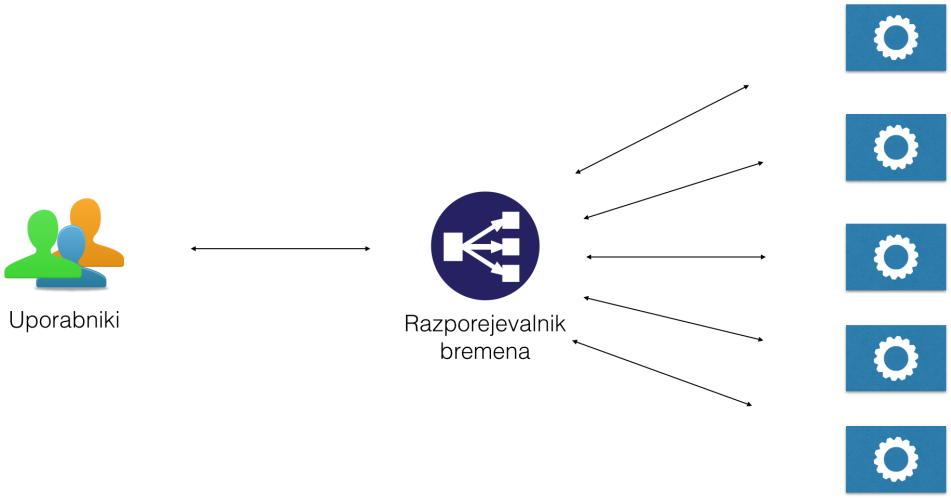
5.1 Načini skaliranja v oblaku

Osredotočili se bomo predvsem na tri načine skaliranja, za katere lahko rečemo, da predstavljajo tri različne dimenzije kocke skalabilnosti (ang. "scale cube"). Izpustili bomo vertikalno skaliranje, saj predstavlja večanje sistemskih virov stroja (več pomnilnika, boljši/več procesorjev, ...), na katerem teče aplikacija. Ko govorimo o večjih obremenitvah, dosežemo maksimalne limite enega stroja zelo hitro, zato omenjeni način v našem kontekstu ne pride v upoštev. Kocka skalabilnosti in opis njenih dimenzij je povzet po [20], ki razdeli kocko na x, y in z osi.

5.1.1 Skaliranje po x-osi

Skaliranje po x-osi je v osnovi navadno horizontalno skaliranje (ang. "scale out"), v katerega postavimo več vzporednih ločenih instanc, med katere razporejamo breme oz. zahteve. Način je preprost in efektiven za večino tipov aplikacije, vključno monolitne, in predstavlja standardni način skaliranja aplikacije tako v zalednih okoljih kot tudi v oblaku. Slika 5.1 prikazuje takšno skaliranje. V osnovi, če imamo N kopij aplikacije, vsaka kopija obdela $\frac{1}{N}$ prometa. Tako skaliramo tudi monolitne aplikacije, ki v veliki večini uporabljajo seje v lokalnem pomnilniku, vendar moramo uporabiti obstojne seje (ang. "sticky sessions"), ki zagotovijo, da bodo zahteve nekega uporabnika vedno pristale na istem strežniku, ki vsebuje njegovo sejo. Kljub fiksnemu usmerjanju posameznih uporabnikov se na dolgi rok breme enakomerno porazdeli.

Ko govorimo o mikrostoritvah, so zadeve še bolj preproste. Ker so v večini primerov mikrostoritve brez stanja, ne potrebujemo obstojnih sej, kar pomeni, da lahko zahteve porazdelimo bodisi naključno bodisi po načinu krožne prioritete (round robin). Ker aplikacijo sestavlja več različnih mikrostoritev, jih lahko poljubno ločeno skaliramo po potrebi. Če uporabimo PaaS okolje, je skaliranje še posebej preprosto. V večini primerov enostavno vnesemo število instanc, ki ga želimo imeti, in platforma poskrbi za zagon in



Slika 5.1: Prikaz skaliranja po x-osi

konfiguracijo. Nekateri ponudniki (npr. Amazon in Heroku) ponujajo storitev, ki nadzoruje promet in glede nanj ustrezno prilagaja število instanc brez ročnega posega.

Skalabilnost dosežemo z uporabo izenačevalnika obremenitve (load balancerja) na četrtem nivoju ISO/OSI modela (TCP), vendar najpogosteje potrebujemo še sedmi nivo (HTTP), ker je potrebno pregledati glavo zahtevka in zagotoviti obstojne seje.

5.1.2 Skaliranje po y-osi

Skaliranje po y-osi je definirano kot funkcijska dekompozicija aplikacije na manjše dele. Povedano pomeni, da zahteve porazdelimo med več instanc glede na zahtevano funkcionalnost. Način je praktično identičen glavni lastnosti samih mikrostoritev, zato se vanj ne bomo spuščali, podrobneje je opisan v poglavju 2.1 in 2.5. Najpogosteje se za ustrezno usmerjevanje zahtevkov na pravilne mikrostoritve uporabi pot v URL-ju (npr. “/katalog” usmeri zahtevo na prvo mikrostoritev, medtem ko “/narocila” usmeri zahtevo na drugo mikrostoritev).

Ko uporabljamo arhitekturo mikrostoritev, v veliki večini primerov upo-

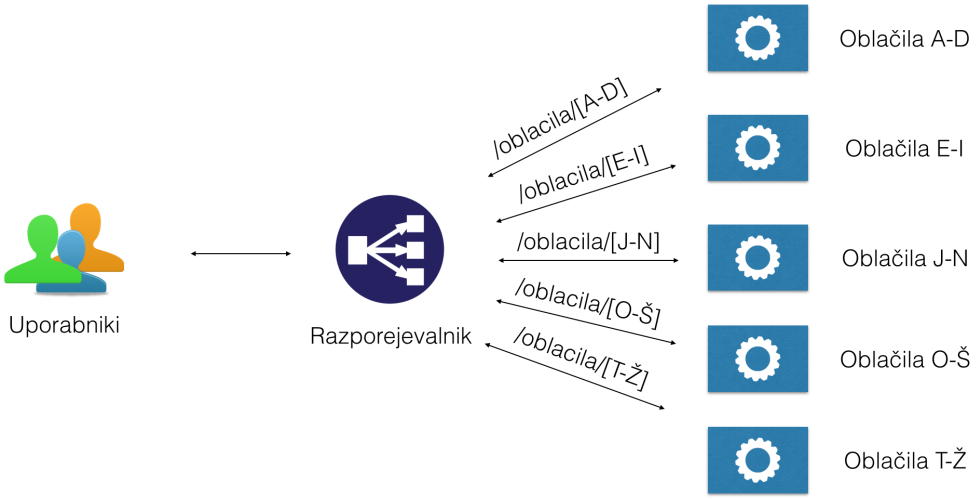
rabljamo kombinacijo x in y skaliranja. Aplikacija je razbita na več funkcijsko zaključenih enot, ki jih posamezno horizontalno skaliramo. Pri PaaS okoljih pomeni, da je vsaka mikrostoritev dodana kot svoja enota, ki ji nato ločeno nastavljamo število instanc in vse preostale konfiguracijske parametre.

5.1.3 Skaliranje po z-osi

Skaliranje po z-osi se nahaja nekje vmes med načinom x in y in deluje po principu črepinjenja (ang. "sharding"), podobno srečamo pri podatkovnih bazah. Pri tem načinu so vse instance enake, podobno kot pri skaliranju po x-osi. Glavna razlika je, da je vsaka instanca odgovorna za neko podmnožico podatkov enakega tipa. Najpogostejši kriterij za razbitje podatkov je primarni ključ entitete, do katere dostopamo (npr. glede na modul pri deljenju primarnega ključa). Drugi najpogostejši kriterij je deljenje glede na tip uporabnika. Npr. plačljive uporabnike preusmerimo na strežnike, ki so manj obremenjeni, hkrati pa zmogljivejši.

Takšno skaliranje je uporabno, ko naša aplikacija vsebuje več nivojev plačljivih politik za dostop do storitev ali podatkov, ki jih ponujamo. Uporabniki, ki plačujejo več, imajo pogosto večja zagotovila za dober prenos podatkov in jih obravnavamo posebej na ločeni lokaciji, kjer imamo bolj sproščena pravila glede povečave števila instanc aplikacije. Za usmerjanje najpogosteje uporabimo poseben tip žetona ali piškotka, da se izognemo zahtevku na podatkovno bazo ob vsakem zahtevku uporabnika. Skaliranje po z-osi prikazuje slika 5.2.

V kontekstu mikrostoritev takšnega skaliranja v večini primerov ne potrebujemo. Zanj bi se odločili le, če je po njem poslovna potreba. V primeru, da potreba obstaja, nam mikrostoritve omogočijo izvedbo z minimalno režijo z razvojem dodatne mikrostoritve, sicer se nam s tem ni treba ukvarjati. Dodatna mikrostoritev bo služila kot usmerjevalnik zahtev na ustrezne instance glede na poslovno zahtevo. V PaaS okoljih postavimo več različnih enot iste glavne mikrostoritve, ki jih nato samostojno konfiguriramo (npr. instance za "zlate" uporabnike so močnejše ter številčnejše).



Slika 5.2: Prikaz skaliranja po z-osi

5.2 Primer skaliranja s pomočjo Docker-ja

Za prikaz primera namestitve in skaliranja aplikacije s pomočjo Docker-ja smo uporabili mikrostoritev [21], ki smo jo razvili v pogavlju 4.3 s pomočjo našega ogrodja. Za demostracijo bomo uporabili odprtokodno okolje Flynn[1], ki skrbi za upravljanje in orkestracijo Docker vsebovalnikov na gruči virtualnih strojev kot tudi ustrezno prevajanje ter poganjanje velikega števila različnih tehnologij. Flynn ponuja odjemalca, preko katerega se lahko na gručo povežemo in namestimo mikrostoritve ter jih enostavno upravljamo.

Za naš testni primer smo namestili gručo treh virtualnih strojev na oblačni infrastrukturi Amazon EC2, tako lahko ustrezno demonstriramo skaliranje naše aplikacije. Flynn v našem primeru deluje kot privaten PaaS oblak. Njegovo delovanje in API-ji so zelo podobni Heroku-ju. Enake korake bi lahko uporabili tudi, če bi mikrostoritev namestili pri njih. Kljub temu Flynn omogoča samo osnovno delovanje PaaS okolja brez naprednih storitev in orodij, ki jih ponujajo komercialni PaaS oblaki. Če bi želeli naprednejše

[1] https://flynn.io

odprtokodno okolje, bi lahko uporabili Deis[2] ali CloudFoundry[3], vendar bo za našo demonstracijo Flynn več kot zadoščal.

Standardna praksa pri večini PaaS oblakov je, da vso konfiguracijo izvedemo s pomočjo ukazne vrstice in Git-a, ki bo poskrbel za prenos kode med nami in oblakom. Večina oblakov sicer ponuja uporabniški vmesnik za upravljanje aplikacij, vendar je za prenos kode še vedno najbolj primeren Git, ki nam ponuja enostavno upravljanje z verzijami aplikacije.

Sprva se premaknemo v imenik, v katerem se nahajata naši mikrostoritvi. Namestili bomo mikrostoritev, ki prikazuje seznam oblačil. Predpostavimo, da imamo na našem računalniku nameščenega odjemalca Flynn za upravljanje in da se z njim povežemo na našo instanco - v tem primeru je naslov slednji "cloud-test.kumuluz.com". Registrirati moramo našo mikrostoritev v ogrodju Flynn, kar prikazuje odsek 5.1.

```
$ flynn create oblacila-katalog
Created example
```

Odsek 5.1: Kreiranje aplikacije na ogrodju Flynn

Ko je ukaz uspešno zaključen, nam je dodeljen avtomatsko zgeneriran URL, ki ga lahko pridobimo z ukazom, ki je prikazan v odseku 5.2.

```
$ flynn route

ROUTE
http:oblacila-katalog.cloud-test.kumuluz.com     ....
```

Odsek 5.2: Poizvedba URL naslova ustvarjenega vnosa mikrostoritve

Preden naložimo našo aplikacijo, je potrebno še specificirati, kako naj se zažene mikrostoritevm kar lahko storimo s pomočjo datoteke "Procfile", ki

[2]https://deis.io
[3]https://www.cloudfoundry.org

jo ustvarimo v korenskem imeniku naše mikrostoritve. V datoteki specificiramo ukaza za poganjanje različnih tipov instanc. Najpogosteje sta spletni poslušalec (sem spada naša mikrostoritev, ki se odziva na HTTP zahtevke) in delavci (napogosteje so instance, ki opravljajo daljša opravila v ozadju, neodvisno od spletnega dela). V našem primeru potrebujemo samo spletnega poslušalca. V datoteko zapišemo isti ukaz, kot smo ga uporabili v poglavju 4.3. Končno stanje datoteke je prikazano v odseku 5.3.

```
web: java -cp katalog/target/classes:katalog/target\
/dependency/* si.fri.ogrodje.ee.EeApplication
```

Odsek 5.3: Vsebina datoteke "Procfile"

Preostalo nam je samo še nalaganje mikrostoritev, kar naredimo s pomočjo Git-a. Prikazuje odsek 5.4.

```
$ git init && git add .
Initialized empty Git repository in ...
$ git commit -m"prvi commit"
...
$ git push flynn master
...
-----> Building oblacila-katalog...
-----> Java app detected
...
-----> Creating release...
=====> Application deployed
=====> Waiting for web job to start...
=====> Default web formation scaled to 1
```

Odsek 5.4: Objava mikrostoritve

Mikrostoritev je bila uspešno nameščena in je na voljo na URL-ju, ki smo ga pridobili iz odseka 5.2. Vidimo lahko, da je okolje avtomatsko zaznalo

javansko aplikacijo in jo ustrezno prevedlo (z uporabo Maven-a), nato pa pognalo z ukazom, ki smo ga specificirali. Če želimo kasneje mikrostoritev posodobiti, lahko ponovno zaženemo ukaz "git push" in okolje bo zgradilo nov vsebovalnik in zamenjalo obstoječega.

V primeru, da uporabljamo kakšno tehnologijo, ki je okolje ne zna prevesti oz. zgenerirati Docker vsebovalnika, lahko podobno kot smo podali "Procfile", podamo "Dockerfile", v katerem specificiramo vse korake, kako zgraditi celoten vsebovalnik. Okolje ga nato zgradi in z njim upravlja tako kot s tistimi, ki jih avtomatsko zgradi samo.

Naslednji korak je, da aplikacijo skaliramo in poženemo več instanc, med katere se bo porazdelilo breme. Za demonstracijo želimo postaviti 3 instance, kar Flynn omogoča z uporabo preprostega ukaza, ki ga prikazuje odsek 5.5.

```
$ flynn scale web=3
scaling web: 1=>3
ID
flynn-3354adffaa7e674db38dd721ed5ba020
flynn-3e8572dd4e5f4136a6a1934eadca5e02
flynn-d55c7a221ef542c123e0feac0892a0b0
```

Odsek 5.5: Skaliranje mikrostoritve

Tečejo tri instance, med katere se bodo zahtevki enakomerno porazdelili. Potrdili smo, da je skaliranje mikrostoritev v PaaS okoljih zelo preprosto; so manjšega obsega in brez stanja, zato jih lahko okolje hitro in poljubno vzpostavlja in uničuje. V večini primerov je dovolj samo, da okolju povemo, koliko instanc želimo, za vse ostalo bo poskrbelo samo. Omenjeno predstavlja velik kontrast tradicionalnemu skaliranju v Javi EE z uporabo aplikacijskih strežnikov, kjer je potrebno ročno namestiti in konfigurirati vse aspekte, kar lahko traja ure, če ne dneve.

Na enak način namestimo še preostale mikrostoritve in jih nato neodvisno in poljubno posodabljamo ter skaliramo. V primeru, da želimo mikrostoritvi

podati konfiguracijske podatke (npr. podatke o podatkovni bazi, naslove drugih mikrostoritev, ...), lahko opisano storimo s pomočjo spremenljivk okolja, ki jih podpirajo vsa PaaS okolja.

Demonstrirali smo, kako enostavna je namestitev in skaliranje mikrostoritev v oblačnem okolju z uporabo našega ogrodja. V primeru, da uporabimo komercialen oblak (npr. Heroku), lahko uporabimo tudi avtomatsko skaliranje glede na obremenitev, ki dodatno olajša upravljanje. Zaključimo lahko, da nam PaaS okolja za skaliranje v oblaku v kombinaciji z mikrostoritvami (in našim ogrodjem za Javo EE) omogočajo enostaven in močen način tako razvoja kot tudi poganjanja visoko zmogljivih skalabilnih in fleksibilnih aplikacij, ki so pripravljene na prihodnost.

Poglavje 6

Sklep

V okviru diplomske naloge je bilo uspešno zasnovano in razvito izvirno ogrodje za razvoj mikrostoritev z uporabo Java EE tehnologij. Ogrodje omogoča razvoj samostojnih in neodvisnih mikrostoritev z uporabo standardnih Java EE tehnologij. Ponudi možnost izbire specifičnih tehnologij v Java EE skladu (npr. JPA, JAX-RS, CDI, ...) in specifičnih implementacij posameznih tehnologij. V primeru, ko neka implementacija ne ustreza oz. vsebuje hrošče ali nima želenih dodatnih funkcionalnosti (npr. Apache CXF za JAX-WS), lahko enostavno izberemo drugo (npr. Metro za JAX-WS) s pomočjo preproste zamenjave odvisnosti v aplikaciji. Ogrodje prav tako avtomatsko konfigurira izbrane komponente in jih zapakira v samostojen arhiv, ki ga nato poženemo v navadnem javanskem izvajalnem okolju. Uspešno smo naslovili problem razvoja mikrostoritev v Javi EE in ponudili skupnosti odprtokodno orodje, ki olajša prehod iz trenutne pogoste monolitne arhitekture v novejšo, moderno arhitekturo mikrostoritev, ki je pripravljena za namestitev in skaliranje v oblačnih infrastrukturah, kar smo prikazali s pomočjo Docker-ja.

Z implementacijo ogrodja in razvojem testnih aplikacij smo demonstrirali arhitekturo mikrostoritev ter njene ključne lastnosti, prednosti in tudi slabosti, predvsem tiste, ki jih poskušamo izboljšati s pomočjo implementiranega ogrodja. Pokazali smo, kako lahko mikrostoritve učinkovito rešijo težave, na katere naletimo pri razvoju današnjih visoko zmogljivih poslov-

nih aplikacij. Predvsem je potrebno narediti dekompozicijo aplikacije na funkcijsko zaključene enote, kar omogoča ogrodje z dobro fleksibilnostjo pri izbiri tehnologij in jezika ter lažjo konfiguracijo namestitve bodisi v oblak bodisi na poljuben način. Izberemo točno to, kar potrebujemo za zadano funkcionalnost in držimo visok nivo efektivnosti. Vse našteto se pri visoko zmogljivih aplikacijah z veliko prometa še kako pozna tako stroškovno kot tudi z vidika uporabniške izkušnje in delovanja. Specifično pri Javi EE je, da ne potrebujemo več aplikacijskega strežnika, čigar uporaba vodi k monolitnim aplikacijam in večjim težavam pri namestitvi v oblak, ampak vsaka mikrostoritev skrbi za svoje izvajalno okolje ter konfiguracijske nastavitve. Dobimo popoln nadzor nad vsako mikrostoritvijo in jo po potrebi ustrezno konfiguriramo. Odprejo se vrata v vedno večji svet PaaS ponudnikov, ki močno olajšajo namestitev mikrostoritev v produkcijo. Ključnega pomena je tudi, da so mikrostoritve brez stanja, kajti v nasprotnem primeru lahko pride do problemov v PaaS okoljih, kjer okolje po potrebi vzpostavlja in uničuje instance mikrostoritve.

Arhitektura mikrostoritev predstavlja potrebno evolucijo obstoječe monolitne arhitekture. S pojavom vedno večjega števila tehnologij ter jezikov in množične selitve aplikacij v oblak ustrezno naslovi potrebe današnjih aplikacij. Kljub precejšnjim razlikam v arhitekturah lahko prehod iz ene v drugo izvedemo postopoma in brez drastičnih sprememb v načinu dela, kot smo demonstrirali z našim ogrodjem. Zaključimo lahko, da ogrodje pomeni pomemben korak k uveljavitvi arhitekture mikrostoritev v ekosistemu Jave EE in omogoča enostaven prehod za obstoječe razvijalce in večje organizacije.

Nadaljnje delo bo vključevalo predvsem dopolnitev mankajočih Java EE tehnologij v prvi verziji ogrodja. Sprva bo poudarek na najbolj popularnih (JSF, JAX-WS, WebSocket), kasneje si želimo dodati še podporo JTA globalnih transakcij in morebitnega EJB vsebovalnika. Sprva bodo dodane referenčne implementacije omenjenih tehnologij, ko bo pokrit celoten spekter Java EE specifikacij, se bomo osredotočili na alternativne implementacije Servlet strežnikov in ostalih specifikacij.

Literatura

[1] Internet users. http://www.internetlivestats.com/internet-users/. Dostopano: 2015-07-31.

[2] Facebook users. http://thenextweb.com/facebook/2014/01/29/facebook-passes-1-23-billion-monthly-active-users-945-million-mobile-users-757-million-daily-users/. Dostopano: 2015-08-28.

[3] Sam Newman. *Building microservices : designing fine-grained systems*. O'Reilly Media, Sebastopol, CA, 2015.

[4] Microservices. http://martinfowler.com/articles/microservices.html. Dostopano: 2015-07-31.

[5] James D. Herbsleb and Rebecca E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, 16(5):63–70, 1999.

[6] Open source at netflix. http://techblog.netflix.com/2012/07/open-source-at-netflix-by-ruslan.html. Dostopano: 2015-07-31.

[7] Netflix and open source. http://www.slideshare.net/adrianco/netflix-and-open-source. Dostopano: 2015-07-31.

[8] Netflix has more than 50 open source projects. http://opensource.com/business/15/3/interview-semmy-purewal-netflix. Dostopano: 2015-07-31.

[9] How we build microservices at karma. https://blog.yourkarma.com/building-microservices-at-karma. Dostopano: 2015-07-31.

[10] How netflix is doing continuous delivery. http://www.slideshare.net/adrianco/flowcon-added-to-for-cmg-keynote-talk-on-how-speed-wins-and-how-netflix-is-doing-continuous-delivery. Dostopano: 2015-07-31.

[11] Marcel Krizevnik and Matjaz B. Juric. Improved soa persistence architectural model. *SIGSOFT Softw. Eng. Notes*, 35(3):1–, May 2010.

[12] Polyglot persistence. http://martinfowler.com/bliki/PolyglotPersistence.html. Dostopano: 2015-07-31.

[13] Starbucks does not use two-phase commit. http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html. Dostopano: 2015-07-31.

[14] Thomas Erl. *SOA: Principles of Service Design*. Prentice Hall, 2007.

[15] Microservices and soa. http://www.oracle.com/technetwork/issue-archive/2015/15-mar/o25architect-2458702.html. Dostopano: 2015-07-31.

[16] Poornachandra Sarang, Frank Jennings, Matjaz Juric, and Ramesh Loganathan. *SOA Approach to Integration: XML, Web services, ESB, and BPEL in real-world SOA projects*. Packt Publishing, 2007.

[17] Node.js at paypal. https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/. Dostopano: 2015-07-31.

[18] Lucas Krause. *Microservices: Patterns and Applications: Designing fine-grained services by applying patterns*. Lucas Krause, 2015.

[19] Adam Bien. *Real World Java EE Patterns Rethinking Best Practices*. Lulu.com, 2009.

[20] Martin Abbott. *The art of scalability : scalable web architecture, processes, and organizations for the modern enterprise.* Addison-Wesley, New York, 2015.

[21] Karl Matthias and Sean P. Kane. *Docker: Up & Running.* O'Reilly Media, 2015.